View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by Publikationsserver der Universität Tübingen

Teilautomatisierte Identifikation von Ursachen von Laufzeitfehlern in Modellen

Dissertation
der Mathematisch-Naturwissenschaftlichen Fakultät
der Eberhard Karls Universität Tübingen
zur Erlangung des Grades eines
Doktors der Naturwissenschaften
(Dr. rer. nat.)

vorgelegt von
M.Sc. Johanna Schneider
aus Duisburg

Tübingen
2016

Gedruckt mit Genehmigung der Mathematisch-Naturwissenschaftlichen Fakultät der Eberhard Karls Universität Tübingen.

Tag der mündlichen Qualifikation:	10.10.2016
Dekan:	Prof. Dr. Wolfgang Rosenstiel
1. Berichterstatter:	Prof. Dr. Herbert Klaeren
2. Berichterstatter:	Prof. Dr. Wolfgang Küchlin

Danksagung

Mein Dank gilt folgenden Personen, die maßgeblich zum Entstehungsprozess dieser Arbeit beigetragen haben:

Prof. Dr. Herbert Klaeren und Prof. Dr. Wolfgang Küchlin haben die Betreuung der Promotion seitens der Eberhard Karls Universität Tübingen übernommen. Die Absprachetreffen in Tübingen haben immer wertvolle Tipps und Anregungen beinhaltet.

Prof. Dr. Klaus-Jörn Lange und Prof. Dr. Michael Kaufmann haben als Prüfer wesentlich zum Abschluss des Promotionsvorhabens beigetragen.

Meine Eltern, Peter und Angela Schneider, haben mir meinen gesamten Ausbildungsweg ermöglicht und mir immer die Chance gegeben, meine eigenen Entscheidungen zu treffen.

Meine Schwester Daniela Rademacher und mein Schwager PD Dr. Andre-

as Rademacher haben für unzählige Motivationsschübe gesorgt. Zusätzlich habe ich diverse Korrekturvorschläge und wertvolles Feedback gerade zu den mathematischen Themen erhalten.

Meine Kollegen vom Team *SMAC* der Daimler AG haben zu unterschiedlichsten Teilthemen ihre Expertise eingebracht. Besonders erwähnen möchte ich an dieser Stelle meinen Teamleiter Dr. Steffen Görzig, der seitens Daimler die Betreuung für diese Promotion übernommen hat. Dabei hatte er immer ein offenes Ohr für meine Fragen und Probleme und konnte sie stets beantworten oder konstruktive Lösungsvorschläge geben.

Meine Freunde haben immer akzeptiert, wenn ich mich phasenweise sehr zurückgezogen habe, um an der Dissertation zu arbeiten. In der gemeinsamen Freizeit habt ihr mich abgelenkt, sodass ich neue Kräfte sammeln konnte.

Allen anderen Personen, die sich hier nicht aufgeführt finden, aber zur Entstehung beigetragen haben, möchte ich ebenfalls danken!

Inhaltsangabe

In der Automobilindustrie gibt es einen stetigen Anstieg an Innovationen. Regelmäßig werden von den Automobilherstellern neue Fahrzeugversionen entwickelt und bereits existierende werden weiterentwickelt. Bei diesen Entwicklungen nimmt die Anzahl an Funktionalitäten zu. Dabei wächst auch der Software-Anteil im Auto und gewinnt immer mehr an Bedeutung. Um die teils sehr komplexen Software-Anwendungen ausreichend abzusichern, werden diverse Analyse-Werkzeuge eingesetzt. Werden bei einer Analyse Fehler identifiziert, so sind deren Ursachen zu korrigieren. Ein großer Anteil des Analyseaufwands, auch Reviewaufwand genannt, wird heutzutage in die Identifikation der Ursachen investiert. Die vorliegende Arbeit befasst sich mit dieser Identifikation, wobei Laufzeitfehler betrachtet werden, die in Software-Anwendungen aufgetreten sind. Es wird ein heuristisches Verfahren präsentiert, welches teilautomatisiert mögliche Ursachen einschränkt. Das Verfahren besteht aus zwei Phasen, einer Lern- und einer Anwendungsphase. In der Lernphase werden Merkmale von bekannten Fehlern aus der Vergangenheit und den entsprechenden Korrektu-

ren erlernt, wie zum Beispiel die Art des Fehlers oder die Art des Operators (mathematische Operationen, Zeigerarithmetik, usw.), in dem der Fehler auftritt. Im Rahmen des Lernprozesses erfolgt eine Gewichtung der Operatoren in Abhängigkeit des jeweiligen Fehlertyps. Die Berechnung der Gewichte resultiert aus einer auf das spezifische Problem angepassten nichtlinearen Optimierung. Dabei werden sämtliche Pfade ermittelt, die zu einem Fehler führen und die darin vorkommenden Operatoren werden dokumentiert. Zum Zeitpunkt der Lernphase sind die Ursachen bekannt, da die Korrekturen der Fehler bereits vorliegen. Diese Kenntnis wird genutzt, um die Pfade in zwei disjunkte Mengen zu unterteilen. Eine Menge beschreibt diejenigen Pfade, die Ursachen enthalten, die zweite Menge besteht aus den verbleibenden Pfaden, also denjenigen, die keine Ursachen enthalten. Diese zwei Mengen dienen als Eingangsdaten für die Optimierung und somit für die Berechnung der Gewichte. Die aus der Lernphase generierten Gewichte werden nachfolgend für die Anwendungsphase genutzt. Zunächst werden erneut alle Pfade ermittelt, die zu einem Fehler führen. Eine Unterscheidung zwischen Ursachen- und Nicht-Ursachenpfaden kann hier jedoch nicht erfolgen, da zu dem Zeitpunkt der Anwendungsphase noch keine Informationen bezüglich der Ursachen vorliegen. Im Anschluss werden die Pfade anhand der in der Lernphase berechneten Operatorgewichte und der Pfadlänge, d.h. der Anzahl der Operatoren in dem jeweiligen Pfad, gewichtet. Dabei gilt die folgenden Prämisse: je höher das Gewicht eines Pfades ist, desto höher ist die Wahrscheinlichkeit, dass dieser eine Ursache für den Fehler enthält. Die Pfade können nun anhand der so entstandenen Priorisierung einem Review unterzogen werden. Für die Evaluierung dieser zweiphasigen Methode werden Software-Modelle im Entwicklungsstadium aus dem PKW-Sektor der Daimler AG verwendet, die mittels der Werkzeuge Simulink von MathWorks bzw. TargetLink von dSPACE entwickelt werden. Es wird gezeigt, dass bei der Anwendung der präsentierten Methode nur 5% aller möglichen Ursachen einem manuellen Review unterzogen

werden müssen. In diesem abschließenden Review können die definitiven Ursachen identifiziert werden, sodass der Aufwand gegenüber dem Erfahrungswert der Experten der Daimler AG signifikant reduziert werden kann.

Inhaltsverzeichnis

Abbildungsverzeichnis ix

Tabellenverzeichnis xi

1 Einleitung 1
1.1 Motivation zu der Lösung des Problems 8
1.2 Aufbau der Dissertation 13

2 Stand der Technik 15

3 Verfahren in der Theorie 23
3.1 Definitionen 23
3.2 Lernphase in der Theorie 28
3.2.1 Lagrange Dualität 38
3.2.2 Subgradientenverfahren 43
3.2.3 Beispiel Lernphase 45
3.3 Anwendungsphase in der Theorie 49
3.3.1 Beispiel Anwendungsphase 51

4 **Verfahren in der Praxis** **55**
4.1 Modellbasierter Entwicklungsprozess 55
4.1.1 Anforderungen 56
4.1.2 Code-Generierung 56
4.1.3 Code-Analyse 57
4.1.4 Ursachenidentifikation 63
4.1.5 Korrektur von Fehlern 64
4.2 Implementierung 65
4.2.1 Übersicht über implementierte Funktionen 66
4.2.2 Vom Modell zum Graphen 67
4.2.3 Implementierung der Lernphase 69
4.2.4 Implementierung der Anwendungsphase 85
4.3 Evaluierung der Ergebnisse 86

5 **Zusammenfassung und weitere Arbeiten** **93**
5.1 Zusammenfassung der Ergebnisse 93
5.2 Vorschläge für weitere Forschungsarbeiten 94

Notationsverzeichnis **97**

Literaturverzeichnis **101**

Abbildungsverzeichnis

1.1 Software-Rückrufe 5
1.2 Zeitliche Verteilung der Rückrufe\Einführung neuer Modelle 6
1.3 Verteilung der Rückrufe (USA) 7
1.4 Anzahl Rückrufe (USA) 8
1.5 Gleichung (1.1) dargestellt in Simulink 10
1.6 Entwicklungsprozess 11
1.7 Zweiphasiges Verfahren 12

3.1 Beispielgraph 25
3.2 Lernphase 28
3.3 Graphenstruktur von Fehler 1 in Abschnitt 3.2.3 46
3.4 Graphenstruktur von Fehler 2 in Abschnitt 3.2.3 47
3.5 Anwendungsphase 50
3.6 Graphenstruktur für Beispiel in Abschnitt 3.3.1 52

4.1 Abbildung 1.5 für Code-Generierung vorbereitet 57
4.2 Abstrakte Interpretation - zu prüfende Punkte 60
4.3 Abstrakte Interpretation - Abstraktion bilden 60
4.4 Polyspace-Analyse - Teil 1 61

4.5 Polyspace-Analyse - Teil 2 62
4.6 Polyspace-Analyse - Teil 3 63
4.7 Polyspace-Analyse - Teil 4 64
4.8 Polyspace-Analyse - Teil 5 65
4.9 Übersicht über geschriebene Matlab-Skripte 67
4.10 Ausschnitt Simulink-Blockbibliothek 68
4.11 Beispielmodell 69
4.12 Graph entstehend aus dem Modell in Abbildung 4.11 70
4.13 Graph für die Tiefensuche 71
4.14 Verwendung des Model-Checkers für Ursachenidentifikation 76
4.15 Subsystem für die Anwendung des Model-Checkers 77
4.16 Prozess zur Identifikation der Fehler 81
4.17 Evaluierung der angewendeten Methode 90
4.18 Beispiel für `Unreachable Code` 91
4.19 Evaluierung für *Unreachable Code* 91

Tabellenverzeichnis

3.1 Anforderungen der Optimierungsmethoden 36
3.2 Mapping der Optimierungsprobleme auf Verfahren 37

4.1 Tiefensuche für Abbildung 4.13 73
4.2 Beispielpfadgewichte 87
4.3 Sortierte Beispielpfadgewichte 87
4.4 Ergebnisse der Evaluierung 89
4.5 Ergebnisse für die Polyspace-Meldung *Unreachable Code* . . 89

1 Einleitung

„Softwär, des weiß jo scho jeder vom Softeis her, isch ebbes Weiches, wo oim, wenn's heiß hergeht, schnell ebbes drneba geha ko. Wenn mr des schnappa will, kriagt mr's net recht in'd Finger und d'Sauerei wird no größer.“ (Albeck und Albeck, 2005)

Dieser schwäbische Spruch besagt, dass Software fehlerhaft sein kann. Die Identifikation und Korrektur dieser Fehler kann sehr aufwändig sein. Zudem werden durch die Korrekturen unter Umständen weitere Fehler erzeugt und die Prozedur beginnt erneut.

Drei Beispiele für bekannte Software-Fehler und ihre Folgen sind der Absturz der Ariane 5, das Therapiegerät namens Therac-25 oder auch das Hamburger Stellwerk der Deutschen Bahn, die nachfolgend beschrieben werden:

- Ariane 5:
  Am 04. Juni 1995 befand sich die Trägerrakete Ariane 5 auf ihrem Jungfernflug. Das verwendete Navigationssystem wurde vom Vorgängermodell Ariane 4 übernommen. Dabei wurde jedoch nicht

oder nicht ausreichend getestet, ob das System für die Ariane 5 geeignet ist. Laut Mandau (2009) konnte eine gemessene Geschwindigkeit in Form einer 64-Bit-Gleitkommazahl von dem System nach dem Start nicht korrekt in einen ganzzahligen 16-Bit-Wert umgerechnet werden, da die 16 Bit dafür nicht ausreichten. Der Fehlerbericht vom Ariane 501 Inquiry Board (1996) besagt, dass die Variablen in der Software der Ariane 4 auf möglichen Überlauf geprüft wurden. Dabei wurden sieben Variablen identifiziert, für die ein Überlauf möglich war. Vier davon wurden durch Einführen von Ausnahmebehandlungen (sogenannte *Exceptions*) abgesichert. Dies wurde für die verbleibenden drei jedoch nicht getan, da sie bei der Ariane 4 entweder physikalisch begrenzt waren oder ein zusätzlicher Sicherheitspuffer verwendet wurde, der nie ausgefüllt wurde. Diese physikalischen Begrenzungen oder der Sicherheitspuffer galten für die Ariane 5 jedoch nicht. Bei der Übernahme der Software für die Ariane 5 wurde dieses nicht beachtet und beschlossen, dass hier keine Änderungen vorgenommen werden. Durch die fehlende Ausnahmebehandlung konnte der Überlauf nicht abgesichert werden und führte zum Absturz der Ariane 5.

- Therac-25:
  Im Jahr 1986 wurde in den USA das Therapiegerät Therac-25 verwendet, um die Bestrahlung von Krebspatienten vorzunehmen. Die Verwendung einer bestimmten Taste bei der Eingabe der Parameter durch den Arzt führte zu der Bildschirmmeldung „Fehlfunktion 54“. Dieser Fehler war jedoch nicht in der Dokumentation beschrieben, sodass das medizinische Personal aufgrund von Unwissenheit den Befehl wiederholte. Dadurch wurden die Patienten überdosiert mit 25 Millionen Elektronen-Volt bestrahlt. DER SPIEGEL (1990) berichtete, dass in Folge dessen zwei Patienten gestorben sind, mindestens drei weitere wurden verletzt.

- Hamburger Stellwerk der Deutschen Bahn:
  Im Jahr 1995 wurde das Stellwerk im Bahnhof Hamburg-Altona durch ein digitales System ersetzt. Nach der Inbetriebnahme stürzte das System alle zehn Minuten ab, sodass das Stellwerk abgeschaltet werden musste. Dies zog ein europaweites Chaos im Bahnverkehr nach sich. Die Fehlerursache lag in diesem Fall an einem zu klein ausgelegten Speicher, sodass es zu einem Datenüberlauf kam. Dieser verursachte die Systemabstürze (vgl. ChannelPartner, 2009).

Auch in der Automobilindustrie gibt es bekannte Software-Fehler:

- BMW:
  Im Jahr 2003 war der thailändische Finanzminister Suchart Jaovisidha mit einem Dienst-BMW auf dem Weg zu einer Konferenz, als das Fahrzeug plötzlich stehen blieb. Die Türen waren verriegelt, die Fenster ließen sich nicht öffnen und auch die Klimaanlage funktionierte nicht mehr. Als Grund wurde ein Fehler im Bordcomputer angegeben. Um den Minister aus seinem Fahrzeug befreien zu können, musste laut Spiegel Online (2003) ein Wachmann eine Scheibe einschlagen.

- Toyota Prius:
  2005 wurde der Toyota Prius aufgrund von zwei Software-Fehlern zurückgerufen. Zum einen verhinderte ein Software-Fehler das Starten des Verbrennungsmotors, weil zu wenig Ansaugluft über die elektronisch geregelte Drosselklappe zugeführt wurde. Zum anderen konnte ein schnelles Durchtreten des Gaspedales als Fehler gesehen werden. „Wird das Gaspedal extrem schnell und kräftig betätigt, besteht die Möglichkeit, dass das Hybrid-Steuergerät ein Signal außerhalb der zulässigen Parameter erhält“, siehe ADAC (2014a). Folglich signalisierte eine Warnlampe den aktivierten Notlauf des Steuergerätes.

- Audi A4:
  Bei Audi gab es 2014 eine Rückrufaktion für den A4. Grund für den Rückruf war ein Fehler bei der Programmierung des Frontairbags. Die Folge war, dass die Airbags möglicherweise nicht auslösen. Von dem Rückruf waren ungefähr 850.000 Fahrzeuge weltweit betroffen, davon 150.000 in Deutschland (vgl. Spiegel Online, 2014).

- Opel Insignia:
  Beim Opel Insignia wurde 2013 ein Kalibrierfehler der Motorkontrolleinheit festgestellt. Eine mögliche Folge wäre ein Motorschaden gewesen. Für ein Software-Update wurden laut der Zeitschrift auto motor und sport (2013) 61.000 Fahrzeuge deutschlandweit zurückgerufen.

- Mercedes-Benz Vito/Viano:
  Eine weitere Rückruf-Aktion gab es 2004 bei Mercedes-Benz für die Modelle Vito und Viano mit Dieselmotoren. Hier wurde ein Software-Fehler im Dieselsteuergerät festgestellt. Dieser Fehler bewirkt, dass die Kraftstoffabschaltung aktiviert wird. Dieses führt zum Ausgehen des Motors. Deutschlandweit waren 3.000 Fahrzeuge betroffen, weltweit insgesamt 10.000 Stück (vgl. heise online, 2004).

In Abbildung 1.1 sind die Anteile der softwarebedingten Rückrufe ausgewählter Automobilherstellern in Deutschland dargestellt. Dazu wurde für die Hersteller Mercedes-Benz, BMW, Audi, VW und Toyota die Rückruf-Statistik des ADAC (2014b) nach Problemen durchsucht, die auf Software-Fehler zurückzuführen sind. Der Betrachtungszeitraum der Statistik erstreckt sich über die Jahre 1996 bis 2014 und zeigt, dass zwischen 5,8% und 21,7% der Rückrufe der betrachteten Hersteller softwarebedingt sind. Die Verteilung der Rückrufe auf die entsprechenden Jahre zeigt Abbildung 1.2. Parallel dazu ist in derselben Abbildung die Anzahl neuer Modelleinführungen der obigen Hersteller dargestellt (vgl. Wikipedia 2014a, 2014b,

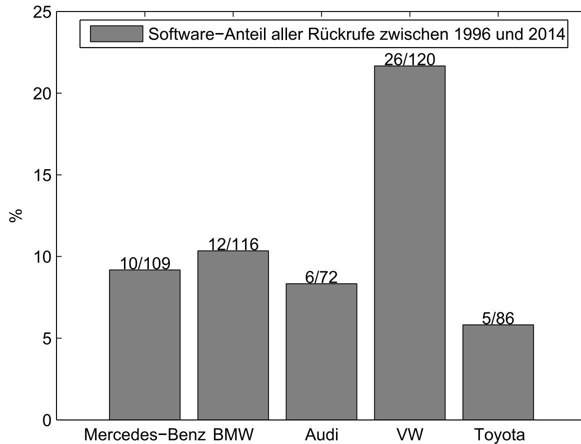

Abbildung 1.1: Anteil aller Rückrufe aufgrund von Software-Problemen aus der ADAC-Rückrufstatistik (ADAC, 2014b).

2014c, 2014d und 2014e). Abgesehen von Ausnahmen ist eine steigende Anzahl der Rückrufe bis 2009 erkennbar und in den Jahren danach eine fallende Anzahl. Die Ursachen für die Rückrufaktionen treten in der Regel nicht unmittelbar nach der Markteinführung auf, sondern mit einem gewissen Zeitabstand. So lassen sich zum Beispiel die Maxima beider Datensätze erklären: 2006 wurden 20 neue Modelle der genannten Hersteller eingeführt, drei Jahre später gab es 15 Rückrufaktionen bezogen auf Software-Probleme. In der Datenbank safercar.gov (2014) sind die Rückrufe in den USA beschrieben, wobei hier sämtliche Hersteller der genannten Quelle berücksichtigt sind. Die zeitliche Verteilung der softwarebedingten Rückrufe ist in Abbildung 1.3 dargestellt. Auch hier lässt sich ein deutlicher Anstieg der Rückrufe erkennen. Die Anzahl der Rückrufe ist nach 2010 auch in den USA wieder gesunken. Zusätzlich zu der Verteilung auf die Zeit ist in Abbildung 1.4 die absolute Anzahl der betroffenen Fahr-

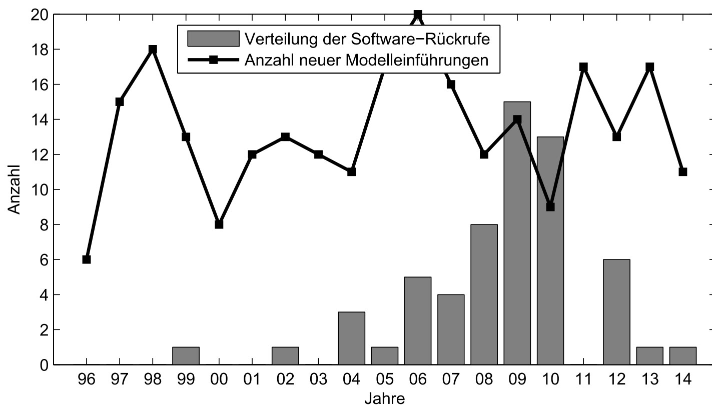

Abbildung 1.2: Zeitliche Verteilung der Rückrufe und Einführung neuer Modelle.

zeuge aufgelistet. Im Jahr 2010 waren zum Beispiel fast 2.500.000 Fahrzeuge von den 240 softwarebedingten Rückrufaktionen betroffen. Da in den Daten vom ADAC (2014b) die Anzahl der jeweiligen Fahrzeuge nicht gegeben ist und auch sonst nicht öffentlich zugänglich ist, kann solch eine Verteilung für die deutschen Rückrufe leider nicht angegeben werden. Mögliche Folgen dieser Rückrufe sind unter anderem zusätzliche Kosten für den Hersteller sowie ein Image-Verlust für den Konzern. Die genannten Effekte lassen sich durch steigenden Umfang an Software in den Fahrzeugen sowie deren Komplexität erklären. Bereits im Jahr 2001 wurden zwischen 50% und 70% der Entwicklungskosten im Bereich Elektronik für Software investiert. Der Software-Umfang verdoppelt sich alle 18 Monate, sodass das Risiko von Software-Fehlern exponentiell ansteigt, siehe INGENIEUR.de (2001). Während 2012 laut der Zeitschrift Automobil Industrie in der Mercedes-Benz E-Klasse 62 Steuergeräte verbaut waren, waren es in rein elektrischen Fahrzeugen bis zu 100 (vgl. Chakraborty et al., 2012). Pro Steuergerät

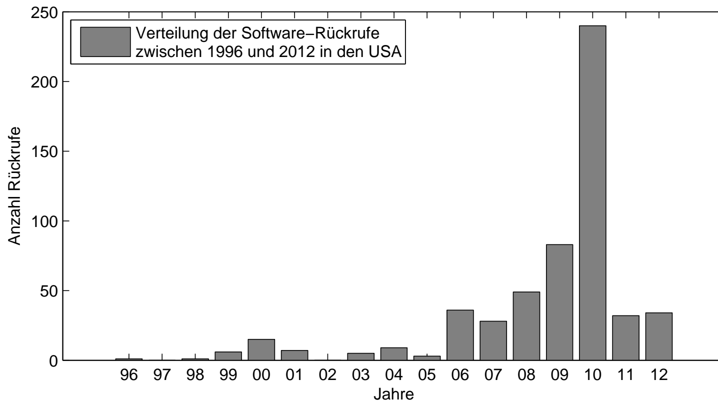

Abbildung 1.3: Zeitliche Verteilung der Rückrufe aus den USA.

nimmt die Anzahl der Code-Zeilen alle 5 Jahre um den Faktor 10 zu. Ein Steuergerät, welches vor 2005 ca. 10.000 Code-Zeilen hatte, wurde 2010 durch ca. 100.000 Zeilen beschrieben (vgl. Werdich, 2013, S. 179). Ein Grund dafür ist auch die steigende Anzahl an Varianten für die Fahrzeuge. Kübler, Zengler und Küchlin (2010) beschreiben, dass geschätzt 10^{21} Konfigurationen für eine Mercedes-Benz C-Klasse und $3 \cdot 10^{24}$ für eine Mercedes-Benz E-Klasse möglich sind. Diese Hardware-Konfigurationen beeinflussen ebenfalls die Software-Entwicklung, da unterschiedliche Hardware-Teile häufig verschiedene Varianten der Software erfordern.

Parallel zu dem enormen Wachstum wurden diverse ISO-Normen aktualisiert, die sich mit dem Thema Software beschäftigen. Eine davon ist die IEC 61508 (IEC - International Electrotechnical Commission), die sich mit funktionaler Sicherheit befasst. Diese Norm wurde im Jahre 2011 als (International Organization for Standardization (ISO), 2012) für die Automobilentwicklung abgeleitet und ist für Fahrzeuge bis 3,5 Tonnen Ge-

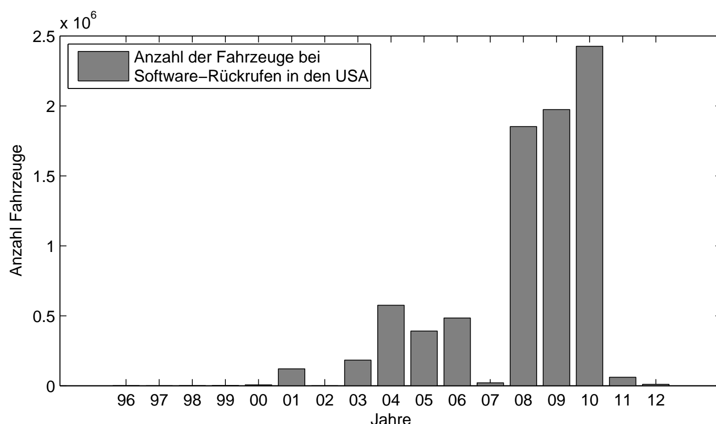

Abbildung 1.4: Anzahl der zurückgerufenen Fahrzeuge in den USA.

wicht gültig. Durch diese Norm wird eine ausreichende Qualitätsicherung vorgeschrieben. Dazu gehören zum Beispiel die Durchführung von statischen Analysen der Software oder die Einhaltung von Programmierrichtlinien, bevor die Software auf das jeweilige Steuergerät geladen wird. So können Laufzeitfehler frühzeitig erkannt und behandelt werden, bevor die betroffenen Autos verkauft werden.

Werden bei der Analyse des Steuergerätecodes Laufzeitfehler gefunden, so müssen zunächst deren Ursachen (siehe Definition 4 im Abschnitt 3.1) gefunden werden, bevor sie korrigiert werden. Die vorliegende Arbeit beschäftigt sich mit der Identifikation dieser Ursachen.

1.1 Motivation zu der Lösung des Problems

Das Ziel der vorliegenden Dissertation ist die Entwicklung einer Methodik, durch deren Einsatz der manuelle Aufwand bei der Identifikation der Ursa-

chen von Laufzeitfehlern in Software reduziert werden kann. Dabei werden Modelle betrachtet, die Fehler enthalten, für welche die Ursachen im Modell identifiziert werden sollen. Conrad et al. (2005) beschreiben *Modelle* als „konstruktive, meist ausführbare Beschreibungen von durch Software zu realisierenden Algorithmen einer Steuerung bzw. Regelung unter Verwendung grafischer Notationselemente.“

Bei der Daimler AG werden für die modellbasierte Entwicklung vorwiegend Simulink und TargetLink verwendet, deswegen ist in dieser Arbeit mit Modell immer ein Simulink-/TargetLink-Modell gemeint. „Simulink ist eine Blockdiagrammumgebung für die Mehrdomämen-Simulation und Model-Based Design. Simulink unterstützt den Entwurf und die Simulation auf Systemebene [...]“, siehe MathWorks (2014c). Diese Umgebung enthält einen Grafikeditor, mit dem via Drag-and-Drop *Blöcke* in das aktuelle Modell eingefügt werden können. Hinter jedem Block stehen Operatoren wie zum Beispiel mathematische Operationen. Die Gleichung

$$x = y + z \tag{1.1}$$

kann in Simulink dargestellt werden anhand von vier Blöcken:

1. einem `Inport`, der den Wert y repräsentiert,
2. einem `Inport`, der den Wert z repräsentiert,
3. einem Additionsblock,
4. einem `Outport` für das Ergebnis x.

Diese Operation ist in Abbildung 1.5 in Simulink dargestellt. Die Linien zwischen den Blöcken signalisieren dabei den Signal- oder Datenfluss. Analog zu Simulink kann auch TargetLink von dSPACE (2014) verwendet werden. Die Blockmenge von TargetLink ist eine Submenge der Simulink-Blockmenge. TargetLink wird in der Regel genau dann verwendet, wenn

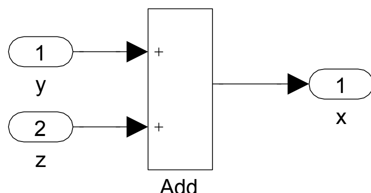

Abbildung 1.5: Gleichung (1.1) dargestellt in Simulink.

weitere Fähigkeiten des Tools, wie zum Beispiel die Code-Generierung, genutzt werden.

Bei dem Review von Steuergeräte-Software in Form von Modellen hat sich gezeigt, dass bestimmte Arten von Laufzeitfehlern wiederholt auftreten. Nicht nur die Art der Fehlers wiederholt sich, sondern auch die Blockstruktur des Modells, die zu diesen Fehlern führt. Aus dieser Beobachtung heraus entstand der Ansatz, eine Methode zu entwickeln, die aus vorangegangenen Fehlern lernt. Für die Entwicklung der Methode gelten die folgenden Ziele, wie sie Klaeren und Sperber (2007, S. 5) angeben: die Lösung des Problems muss konstruktiv und effizient sein. Die Konstruktivität wird in dieser Arbeit durch die Anwendbarkeit der neuen Methode respräsentiert, die Effizienz durch den reduzierten Aufwand für den Anwender.

Der etablierte Prozess während der Steuergeräteentwicklung ist in Abbildung 1.6 dargestellt. Ausgehend von gegebenen Anforderungen kann ein Modell erstellt werden. Dabei kann es vorkommen, dass sich Anforderungen ändern oder neue Anforderungen dazu kommen, sodass das Modell geändert werden muss. Für das erstellte Modell kann Code generiert werden, der im nächsten Schritt analysiert wird. Findet die Analyse keine Fehler, so ist der Prozess an der Stelle beendet. Werden jedoch Fehler er-

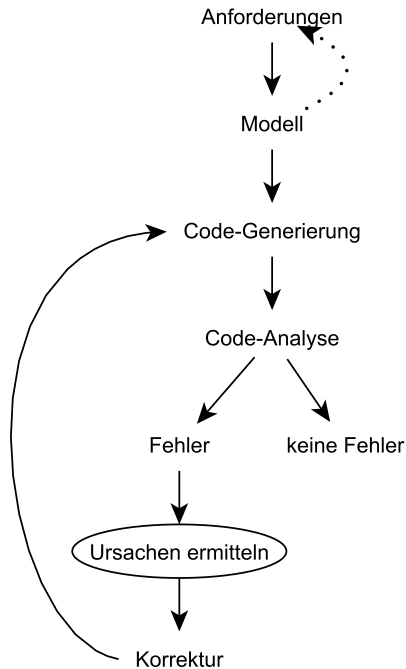

Abbildung 1.6: Entwicklungsprozess für Steuergeräte.

mittelt, so müssen deren Ursachen ermittelt und anschließend korrigiert werden. Nach der Korrektur wird erneut Code generiert und analysiert. Die erneute Analyse erfolgt, um zu prüfen, ob der Fehler tatsächlich korrigiert wurde und ob die Korrektur möglicherweise neue Fehler verursacht hat.

Die in dieser Arbeit präsentierte Methode bezieht sich auf den Prozessschritt der Ursachenidentifikation. Diese Methode besteht aus zwei Pha-

sen, einer Lern- und einer Anwendungsphase (vgl. Abbildung 1.7). Ausgangspunkt für die Lernphase ist eine Menge an Fehlern. Das Ergebnis der Lernphase wird an die Anwendungsphase übergeben. Informationen, die aus der Anwendungsphase resultieren, werden zurück an die Lernphase übermittelt, um diese bei Erfolg der Anwendungsphase zu verfeinern oder bei ausbleibendem Erfolg entsprechende Korrekturen vorzunehmen. Ein Beispiel für einen ausbleibenden Erfolg ist eine Überapproximation, bei der zu viele Daten als Ursachen markiert werden. Dazu kann eine weitere Iteration der Lernphase durchgeführt werden, welche zusätzlich zu den vorherigen Daten die Information der Überapproximation erhält.

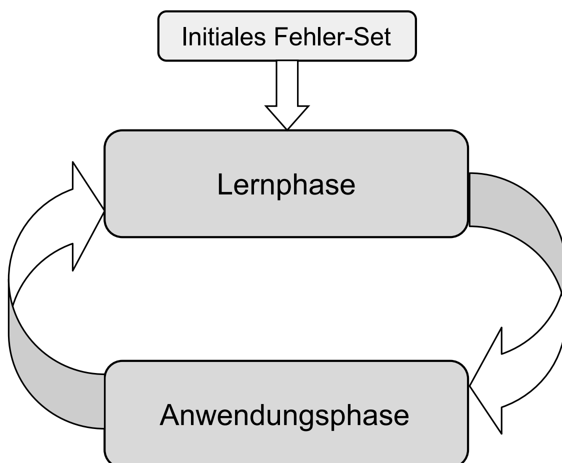

Abbildung 1.7: Zweiphasiges Verfahren.

Die Lernphase beginnt mit einer Fehlermenge, wobei die Ursachen der jeweiligen Fehler bekannt sind. Im ersten Schritt werden alle Pfade be-

trachtet, die zum jeweiligen Fehler führen. Diese Pfade werden in zwei disjunkte Mengen eingeteilt. Die erste Menge enthält die Pfade, die Ursachen enthalten. Die zweite Menge beinhaltet die verbleibenden Pfade. Die beiden Pfadmengen sollen mathematisch unterscheidbar sein, daher werden Pfadgewichte eingeführt. Ein Pfadgewicht berechnet sich aus den im Pfad enthaltenen Operatoren. Dafür werden die Operatoren selbst gewichtet. Diese Operatorgewichtung wird durch eine nichtlineare Optimierung berechnet und dient als Eingabe für die Anwendungsphase. Auch hier werden wieder alle Pfade ermittelt, die zum aktuellen Fehler führen. Eine Unterteilung in disjunkte Mengen ist zu dem Zeitpunkt nicht möglich, da hier erst ermittelt wird, welche Pfade Ursachen enthalten. Durch die berechneten Operatorgewichte können die Pfadgewichte berechnet werden. Das Ergebnis der Anwendungsphase ist ein Ranking aller Pfade nach absteigendem Pfadgewicht. Dabei soll gelten: je weiter vorne ein Pfad im Ranking steht, desto höher ist die Wahrscheinlichkeit, dass er die Ursache enthält.

1.2 Aufbau der Dissertation

Diese Arbeit ist wie folgt aufgebaut: in Kapitel 2 werden verwandte Arbeiten sowie der Unterschied zwischen diesen und dem Fokus der vorliegenden Arbeit beschrieben. Anschließend werden in Kapitel 3 die theoretischen Schwerpunkte des Verfahrens erläutert. Dieses Kapitel ist aufgeteilt in notwendige Definitionen und die Beschreibung der Anwendungs- und der Lernphase. Für beide Phasen wird zusätzlich ein Beispiel präsentiert. Die Theorie wird in Kapitel 4 auf Daten aus der PKW-Entwicklung der Daimler AG angewendet. In diesem Kapitel wird dargestellt, wie das Verfahren implementiert wird, welche Werkzeuge verwendet werden und welche Ergebnisse das Verfahren ergibt. Zusammengefasst wird diese Arbeit in Kapitel 5.

2 Stand der Technik

Im vorherigen Kapitel wurde gezeigt, dass der Software-Anteil im alltäglichen Umfeld immer größer wird. Aus diesem Grund wurden diverse Werkzeuge entwickelt, die Software bezüglich einer Vielzahl von Schwachstellen analysieren können. Zum jetzigen Zeitpunkt geben diese Werkzeuge jedoch keine Ursachen für diese Schwachstellen aus. In der Regel werden nur die Pfade aufgezeigt, die zu der jeweiligen Schwachstelle führen, diese werden aber nicht bzgl. der Wahrscheinlichkeit bewertet, die Ursache für diesen Fehler zu enthalten.

Zeller (2006) beschreibt eine Methode zur Identifikation von Ursachen, die im Rahmen vom Testen angewendet werden kann. Dabei wird eine Ursache als Differenz zwischen zwei Welten erklärt (vgl. Zeller, 2006, S. 318-323):

- eine Welt, in welcher der Fehler auftritt und
- einer alternativen Welt, in welcher der Fehler *nicht* auftritt

Die alternative Welt soll der tatsächlichen Welt so ähnlich wie möglich sein, also nur minimal verändert. Die tatsächliche Ursache wird als die minimale Differenz zwischen diesen beiden Welten angegeben. Sollten mehrere Ursachen möglich sein, dann soll diejenige gewählt werden, deren alternative Welt der tatsächlichen Welt am ähnlichsten ist. Angewendet auf das Testen ergibt dies die folgende Strategie: es wird ein Testfall generiert, der sukzessive verändert wird. Dabei wird nur eine Änderung pro Schritt vorgenommen. Sobald der Fehler auftritt, ist die Ursache als Differenz zwischen diesem und dem vorangehenden Testfall identifiziert (vgl. Zeller, 2006, S. 331).

Jones und Harrold (2005) verwenden in dem Werkzeug *Tarantula* ebenfalls die Testingmethode, um Ursachen einzuschränken. Dabei werden Informationen verwendet, die durch das Testen bereits gegeben sind. Die primäre Information dabei ist, ob ein Testfall fehlschlägt oder erfolgreich ist. Die Idee der Methode ist es, dass jede Operation im Code mittels eines Farbspektrums von rot über gelb zu grün eingefärbt wird:

- Operationen, die hauptsächlich durch fehlschlagende Testfälle abgedeckt werden, werden rot eingefärbt,
- Operationen, die hauptsächlich durch erfolgreiche Testfälle abgedeckt werden, werden grün eingefärbt und
- Operationen, die sowohl durch fehlschlagende als auch erfolgreiche Testfälle abgedeckt werden, werden gelb eingefärbt.

Die Farbe einer Operation o wird dabei durch die Formel

$$farbe(o) = \frac{\frac{erfolgreich(o)}{gesamterfolgreich}}{\frac{erfolgreich(o)}{gesamterfolgreich} + \frac{fehlgeschlagen(o)}{gesamtfehlgeschlagen}} \tag{2.1}$$

bestimmt. Dabei ist $erfolgreich(o)$ die Anzahl der Testfälle, welche die Operation o erfolgreich durchlaufen sind. Die Variable $gesamterfolgreich$ gibt die Anzahl der erfolgreichen Testfälle an. Analog $fehlgeschlagen(o)$ und $gesamtfehlgeschlagen$ mit fehlgeschlagenen Testfällen. Die Funktion $farbe(o)$ befindet sich im Intervall $[0, 1]$, wobei eine Operation o_1 mit $farbe(o_1) \to 0$ mit höherer Wahrscheinlichkeit eine Ursache beschreibt als eine Operation o_2 mit $farbe(o_2) \to 1$.

Ein ähnlicher Ansatz wird durch Abreu et al. (2007) verwendet. Ein wesentlicher Unterschied zu dem Tarantula-Werkzeug ist die Verwendung eines anderen Ähnlichkeitskoeffizienten. Anstelle von (2.1) wird hier der sogenannte Ochiai-Koeffizient verwendet:

$$s(o) = \frac{erfolgreich(o)}{\sqrt{gesamterfolgreich * (erfolgreich(o) + fehlgeschlagen(o))}}. \tag{2.2}$$

Reder und Egyed (2013) beschreiben die Suche nach Ursachen von *design model inconsistencies*, d.h. nach Ursachen für Verletzungen von Design-Regeln. Dazu wird die Methode des SAT-Solvings verwendet. Dieses Verfahren beruht auf der Bestimmung aller MUS (minimal unsatisfied subsets of constraints). Die Vereinigung aller dieser MUS ist die Ursache für die Unerfüllbarkeit der Design-Regeln.

Die Methode des Model-Checkings liefert ebenfalls Ansätze für die Identifikation von Ursachen. Baier und Katoen (2008, S. 8) beschreiben Model-Checking als eine Verifikationstechnik, bei der sämtliche Systemzustände ermittelt werden. Diese Methode kann mit einem Programm für Computer-Schach verglichen werden, welches alle möglichen Züge systematisch berechnet. Beim Model-Checking muss eine Eigenschaft definiert werden, die durch den Model-Checker analysiert werden soll. Kann der Model-Checker beweisen, dass diese Eigenschaft eintreten kann, so liefert er eine sogenann-

te *Trace* zurück, d.h. einen Pfad zu einem Zustand, in dem die Eigenschaft gilt, und der Hinweise auf die Ursache geben kann. Diese Methode wird jedoch dadurch beschränkt, dass aktuell keine große Software analysiert werden kann. Dabei tritt das durch Barnat et al. (2013) beschriebene Problem der *State Space Explosion* auf, d.h. zu viele Zustände, die nicht effizient bearbeitet werden können. Ebenfalls kann die zu analysierende Eigenschaft schwierig zu formulieren sein, wodurch die Wahrscheinlichkeit erhöht wird, dass die Formulierung selbst fehlerbehaftet ist.

Schrammel, Kroening, Brain, Martins, Teige und Bienmüller (2014) verwenden einen Bounded Model-Checker (BMC) für eingebettete Systeme aus dem Automotive-Bereich, genauer gesagt für C-Code, der aus Simulink-Modellen generiert wurde. Diese Art des Model-Checkings wird deswegen als *Bounded* Model-Checking bezeichnet, da die in der Software enthaltenen Schleifen nur bis zu einer fixen Zahl k ausgerollt werden. Zwei Anwendungsfälle werden in der verwiesenen Arbeit beschrieben: Verifizierung und Testen. Dabei gilt, dass die zu verifizierende bzw. zu testende Eigenschaft auf ein Erreichbarkeitsproblem in der Software reduziert werden kann, d.h. auf ein SAT-Problem. BMC kann die Erreichbarkeit eines Zustandes beweisen, während die Unerreichbarkeit durch Induktion über k gezeigt werden kann. Wird eine Reihe von ähnlichen SAT-Problemen bearbeitet, so sollen durch *inkrementelles SAT-Solving* unter Verwendung von Annahmen Informationen und interne Zustände von vorherigen Instanzen wiederverwendet werden. Angewendet wird das inkrementelle SAT-Solving in Kombination mit dem BTC EmbeddedTester und dem BTC EmbeddedValidator, beides Werkzeuge der Firma BTC Embedded Systems AG. Dazu wird durch eine Master-Routine das nächste Verifikations-/Testziel ausgewählt. Die Software wird durch Methoden wie *Slicing* (für das Ziel nicht relevante Software wird vernachlässigt) oder das Ausrollen der Schleifen vorverarbeitet. Das Ergebnis der Vorbereitung wird zusammen mit

dem Ziel an den BMC übergeben und verarbeitet. Dabei entstehen zwei mögliche Situationen:

- der BMC kann das Problem innerhalb einer vom Anwender vorgegebenen Zeit lösen. Ausgegeben wird die Information über Unerreichbarkeit bzw. Erreichbarkeit. Im letzteren Fall gibt es ein Gegenbeispiel bzw. einen Testfall für das Verifikations-/Testziel.
- der BMC kann das Problem nicht lösen, da es vorab zu einem Timeout kommt. Dennoch werden Informationen über bisher gelöste ausgerollte Schleifen an den Anwender ausgegeben.

Im Falle eines Gegenbeispiels bzw. eines Testfalls ist ein möglicher Ursachenpfad gefunden. Es ist hier jedoch nicht bewertbar, wie viele weitere Pfade noch möglich sind. Außerdem wird das Gegenbeispiel bzw. der Testfall für den generierten Code ausgegeben und nicht für das Modell selbst.

Chen et al. (2004) präsentieren einen Ansatz zur Ermittlung von Ursachen von Fehlern in großen Internetseiten. Durch ein Training von Entscheidungsbäumen (*decision trees*) werden Systemkomponenten identifiziert, die mit dem jeweiligen Fehler korrelieren.

Ein weiterer Ansatz zur Ursachenidentifizierung ist durch Hangal und Lam (2002) gegeben. Diese Java-basierte Methode ermittelt dynamisch alle Invarianten und identifiziert parallel, ob es eine Verletzung der bisher gefundenen Invarianten gibt. Das Verfahren ist so konzipiert, dass es kontinuierlich während der Software-Ausführung aktiv ist.

Julisch (2003) betrachtet das Thema Angriffserkennungssysteme. Bei einem Angriff auf die Software wird ein Alarm ausgelöst. Um die Ursachen für die Alarme identifizieren zu können, werden die Angriffsalarme gruppiert, sodass alle Alarme einer Gruppe dieselbe Ursache haben. Die Idee

dabei: wird eine Ursache einer Gruppe verstanden, so werden automatisch alle Ursachen dieser Gruppe verstanden.

Liu, Lucia, Nejati und Briand präsentierten im Oktober 2015 ein Verfahren, welches durch die Kombination von statistischem *Debugging* und dynamischen *Model Slicing* Fehler in Simulink-Modellen lokalisiert. Als Eingabe für das Verfahren dient ein Simulink-Modell M, eine Testsuite $TS = \{tc_0, \ldots, tc_n\}$ und ein Testorakel $\mathcal{O}$. Innerhalb von M sei O die Menge aller `Outport`-Blöcke auf allen Ebenen des Modells, d.h. sowohl im Modell selbst als auch innerhalb der Subsysteme. Dabei gilt, dass $\mathcal{O}$ ermittelt, ob jeder `Outport` $o \in O$ den Testfall tc_i für alle Testfälle $tc_i \in TS$ erfolgreich durchläuft oder ob er fehlschlägt. Das Ziel des Verfahrens ist ein Ranking über alle Blöcke im Modell, wobei die das Ranking anführende Blöcke am wahrscheinlichsten fehlerhaft sind. Dieses Ziel wird durch drei Schritte erreicht:

- Ausführung der Testfälle: das Modell M wird für jeden Testfall ausgeführt. Dabei werden die Informationen gespeichert, welche Testfälle für welchen `Outport` erfolgreich sind bzw. fehlschlagen. Zusätzlich wird eine Liste ausgegeben, welche die Modell-Abdeckung (*coverage*) für die jeweiligen Testfälle beinhaltet. Für jeden Testfall zählt diese Liste die Simulink-Blöcke auf, die durch den Testfall durchlaufen werden.

- Slicing: zunächst wird eine Rückwärtssuche ausgehend von einem `Outport` durchgeführt. Die Suche endet, wenn ein Block vom Typ `Inport` oder `Constant` erreicht wird. Falls Schleifen im Modell vorhanden sind, so werden sie nur einmalig durchlaufen. Die Schnittmenge dieser Blöcke mit denen durch die Testfälle abgedeckten Blöcke wird als *Slice* weiterverarbeitet.

- Ranking: die im vorherigen Schritt identifizierten Slices werden nun

gewichtet. Dazu werden verschiedene Koeffizienten vorgeschlagen wie zum Beispiel Tarantula oder Ochiai. Diese werden mit $Score_o(b)$ bezeichnet für einen `Outport` o und einen Block b. Abhängig von der Wahl der Berechnungsvorschrift kann ein Wert $Score$ für jeden Block b im Modell berechnet werden:

$$Score(b) = \frac{\sum\limits_{\substack{o \in O, \\ Score_o(b) \neq NaN}} Score_o(b)}{|\{o \in O | Score_o(b) \neq NaN\}|}.$$

Als nächstes werden alle Blöcke mit dem gleichen $Score$ gruppiert. Jede der Ranking-Gruppen erhält dabei jeweils einen minimalen und einen maximalen Wert. Ersterer beschreibt die minimale Anzahl an Blöcken, die geprüft müssen, wenn der fehlerhafte Block als erstes geprüft wird. Analog zeigt der zweite Wert die maximale Anzahl an Simulink-Blöcken, die geprüft werden müssen, falls der fehlerhafte Block als letztes geprüft wird.

Die Autoren vergleichen ihre Methode mit dem in dieser Dissertation aufgezeigten Verfahren. [1] Auch wenn die Ansätze einige Parallelen enthalten, so weisen die Methoden inhaltliche Unterschiede aus:

- Der Ansatz von Liu et al. beruht auf Testing, während er in der vorliegenden Dissertation auf statischer Analyse aufbaut.
- Die betrachteten Fehler unterscheiden sich voneinander. Liu et al. bearbeiten drei Arten von Fehlern:
    1. falsche Funktion: Fehler in einer Blockfunktion wie zum Beispiel „>“ anstelle von „≥“,
    2. falsche Verbindung: Fehler in Signallinie zwischen zwei Blöcken wie zum Beispiel Vertauschung der Eingänge in einen Block und

[1] Das in dieser Dissertation beschriebene Verfahren wurde 2014 im Rahmen einer Konferenz vorgestellt, siehe Schneider (2014).

3. falscher Wert: Fehler im Parameter eines konstanten Blockes.

Die vorliegende Arbeit hingehen betrachtet Laufzeitfehler, welche in dem generierten Code von Modellen gefunden werden und deren Ursachen im Modell selbst identifiziert werden sollen.

Mit Ausnahme von Liu et al. (2015) und im Ansatz Schrammel et al. (2014) wird in keiner der vorgestellten Forschungsarbeiten Software in Form von Modellen behandelt. Nur durch Chen et al. (2004) wird eine Lernmethode verwendet. Hier werden jedoch Entscheidungsbäume trainiert und keine Software-Pfade, wie es in der vorliegenden Arbeit der Fall ist. Keines der aufgezählten Verfahren identifiziert die Fehler, für welche die Ursachen berechnet werden sollen, durch das Prinzip der abstrakten Interpretation. Ein weiterer Unterschied zwischen den genannten Verfahren und der Methode dieser Arbeit ist die Art der analysierten Fehler: in der vorliegenden Arbeit werden Laufzeitfehler behandelt, während in den aufgezählten größtenteils funktionale Fehler betrachtet werden.

Zum jetzigen Zeitpunkt ist der Autorin kein Verfahren bekannt, welches mit der in dieser Arbeit präsentierten Methode vergleichbar ist.

3 Verfahren in der Theorie

Die in dieser Arbeit vorliegende Ausgangssituation kann folgendermaßen beschrieben werden: gegeben sei eine Software mit bekannten Fehlern, die korrigiert werden sollen. Bevor jedoch eine solche Korrektur erfolgen kann, muss zunächst ermittelt werden, wodurch diese Fehler auslöst werden. Gesucht sind also die *Ursachen* der Fehler. Die in der Arbeit entwickelte Ursachensuche besteht aus zwei Phasen, einer Lern- und einer Anwendungsphase.

3.1 Definitionen

Bevor das Verfahren vorgestellt wird, werden an dieser Stelle einige Begriffe erläutert, die für die nachfolgenden Abschnitte notwendig sind.

Definition 1 (Vektor, Matrix)**.**
Fett gedruckte Buchstaben in dieser Arbeit bezeichnen *Vektoren* oder *Matrizen*. Ist die Kennzeichnung durch einen Kleinbuchstaben gegeben, so handelt es sich um einen *Vektor*. Eine *Matrix* wird durch einen fettgedruckten Großbuchstaben gekennzeichnet.

Definition 2 (Norm)**.**
Sei $\mathbf{x} \in \mathbb{R}^n$ ein Vektor. Dann bezeichnet $||\mathbf{x}|| = ||\mathbf{x}||_2 = \sqrt{\sum_{i=1}^{n} x_i^2}$ die *Euklidische Norm* von $\mathbf{x}$.

Definition 3 (Laufzeitfehler)**.**
Ein *Laufzeitfehler* ist ein Fehler, der während der Laufzeit, also der Ausführung einer Software, auftritt. Kästner et al. (2009) beschreiben einen Laufzeitfehler wie folgt: „Laufzeitfehler führen in der Regel dazu, dass nach ihrem Auftreten das Programmverhalten undefiniert ist. Einige Laufzeitfehler, z.B. floatingpoint Überläufe, können Interrupts auslösen und somit abgefangen werden - vorausgesetzt, ein entsprechender Interrupthandler wurde implementiert. Die möglichen Folgen von Laufzeitfehlern wie beispielsweise Feldgrenzenverletzungen beim Zugriff auf Feldelemente (array access out-of-bounds) oder die Dereferenzierung eines ungültigen Zeigers reichen von fehlerhaftem Programmverhalten bis hin zum Softwareabsturz.“

Definition 4 (Ursache)**.**
Als *Ursache* wird diejenige Software-Stelle bezeichnet, die für die Entstehung eines Laufzeitfehlers verantwortlich ist. In der Literatur wird dieses auch als *Defekt* bezeichnet, unter anderem durch Zeller (2006).

Definition 5 (Graph)**.**
Ein *Digraph*, vereinfacht auch *Graph* genannt, $G = (E, K)$ besteht aus einer Menge an *Ecken* (*Knoten*)

$$E = \{e_1, e_2, \ldots, e_n\}$$

und einer Menge an gerichteten *Kanten*

$$K = \{(e_i, e_j) \mid e_i, e_j \in E\},$$

wobei e_i der Startknoten und e_j der Zielknoten ist. Ein Beispielgraph ist in Abbildung 3.1 dargestellt. Hier gilt

$$E = \{1, 2, 3, 4\}$$

und

$$K = \{(1,3), (2,1), (2,3), (2,4), (4,2)\}\,.$$

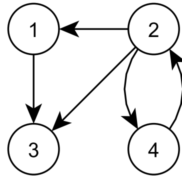

Abbildung 3.1: Beispielgraph.

Definition 6 (Pfad)**.**
Ein *Pfad* p in einem Graphen $G = (E, K)$ ist eine Folge von Knoten

$$p = e_1 e_2 \dots e_k$$

mit $k \in \mathbb{N}$, $e_i \in E$ und $(e_i, e_{i+1}) \in K$ für $1 \leq i \leq k-1$. Ein Pfad in Abbildung 3.1 ist zum Beispiel $4 \to 2 \to 1 \to 3$.

Definition 7 (False Positive)**.**
Bei der Überprüfung einer Eigenschaft können vier Situationen entstehen. Angenommen, es soll geprüft werden, ob die Eigenschaft x wahr sein kann. Die vier resultierenden Fälle lauten:

1. *true positive*: die Prüfung fällt positiv aus und x ist wahr,

2. *false positive*: die Prüfung fällt positiv aus, aber x ist nicht wahr,

3. *false negative*: die Prüfung fällt negativ aus, aber x ist wahr,

4. *true negative*: die Prüfung fällt negativ aus und x ist nicht wahr.

In den Kapiteln 3.2.1 und 3.2.2 werden zwei nichtlineare Optimierungsverfahren beschrieben, für die weitere spezifische Definitionen benötigt werden:

Definition 8 (Konvexe Menge)**.**
Eine Menge $D \subset \mathbb{R}^n$ heißt *konvex* genau dann, wenn

$$\{(1-t)\cdot x + t\cdot y \mid 0 \leq t \leq 1\} \subset D \; \forall \; x, y \in D.$$

Definition 9 (Konvexe Funktion)**.**
Eine Funktion $f : D \to \mathbb{R}$, $D \subset \mathbb{R}^n$, heißt *konvex* auf $C \subset D$, wenn C nichtleer und konvex ist und die Bedingung

$$f((1-t)\cdot x + t\cdot y) \leq (1-t)\cdot f(x) + t\cdot f(y)$$

für alle $x, y \in C$ und $t \in [0,1]$ erfüllt ist (vgl. Alt, 2002, S. 7).

Definition 10 (Konvexe Hülle)**.**
Die *konvexe Hülle* $\mathrm{conv}(X)$ einer nichtleeren Menge $X \subset \mathbb{R}^n$ ist die kleinste konvexe Teilmenge des $\mathbb{R}^n$, welche die Menge X noch enthält. Somit gilt $X \subseteq \mathrm{conv}(X)$. Ist X konvex, so gilt $X = \mathrm{conv}(X)$, vgl. Geiger und Kanzow (2002, S. 18).

Definition 11 (Relatives Inneres)**.**
Es bezeichne $\mathrm{aff}(X)$ die affine Hülle der Menge $X \subset \mathbb{R}^n$, also den Durchschnitt aller affinen Unterräume des $\mathbb{R}^n$, welche X als Teilmenge enthalten. Ein Punkt $\mathbf{x} \in X$ gehört dann zum *relativen Inneren* von X, wenn es eine Kugelumgebung von $\mathbf{x} \in \mathbb{R}^n$ gibt, deren Durchschnitt mit $\mathrm{aff}(X)$ eine Teilmenge von X ist (vgl. Geiger und Kanzow, 2002, S. 322).

Definition 12 (Projektion)**.**
Sei $X \subseteq \mathbb{R}^n$ nichtleer, abgeschlossen und konvex. Dann existiert zu jedem $\mathbf{y} \in \mathbb{R}^n$ ein eindeutig bestimmter Vektor $\mathbf{z} \in X$ mit

$$||\mathbf{y} - \mathbf{z}|| \leq ||\mathbf{y} - \mathbf{x}|| \; \forall \; \mathbf{x} \in X.$$

Der Vektor $\mathbf{z}$ heißt *Projektion* von $\mathbf{y}$ auf X und wird mit $\text{Proj}_X(\mathbf{y})$ bezeichnet (vgl. Geiger und Kanzow, 2002, S. 28).

Definition 13 (Aktive Ungleichungsrestriktion)**.**
Gegeben sei ein Optimierungsproblem der Struktur

$$\min\{f(\mathbf{x}) \mid \mathbf{x} \in \mathbb{R}^n, \mathbf{g}(\mathbf{x}) \leq \mathbf{0}\}$$

mit $\mathbf{g} : \mathbb{R}^n \to \mathbb{R}^{2n}$, $\mathbf{g} = (g_1(\mathbf{x}), \ldots, g_{2n}(\mathbf{x}))$. Dann bezeichnet

$$I(\mathbf{x}) = \{i \mid g_i(\mathbf{x}) = 0\}$$

die Menge der *aktiven Ungleichungsrestriktionen* der Funktion $\mathbf{g}$ (vgl. Geiger und Kanzow, 2002, S. 44).

Definition 14 (Subgradient, Subdifferential)**.**
Sei $X \subseteq \mathbb{R}^n$ offen und konvex, $f : X \to \mathbb{R}$ eine konvexe Funktion und $\mathbf{x} \in X$. Ein Vektor $\mathbf{s} \in \mathbb{R}^n$ heißt *Subgradient* von f in $\mathbf{x}$, wenn gilt:

$$f(\mathbf{y}) \geq f(\mathbf{x}) + \mathbf{s}^\top(\mathbf{y} - \mathbf{x}) \; \forall \; \mathbf{y} \in X.$$

Die Menge der Subgradienten von f in $\mathbf{x}$ heißt *(konvexes) Subdifferential* von f in $\mathbf{x}$ und wird mit $\partial f(\mathbf{x})$ bezeichnet (vgl. Geiger und Kanzow, 2002, S. 330).

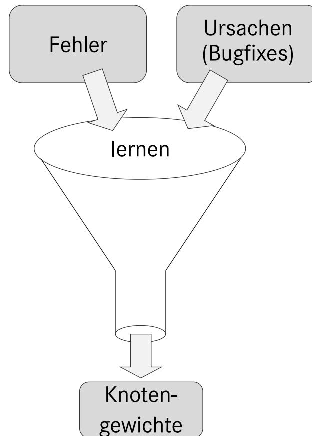

Abbildung 3.2: Lernphase.

3.2 Lernphase in der Theorie

Die Lernphase beruht auf den Erkenntnissen aus vorangehenden Software-Analysen. Bei der Betrachtung diverser Software hat sich gezeigt, dass gewisse Fehlerarten wiederholt auftreten. Diese Feststellung soll bei der Ursachenermittlung genutzt werden. Abbildung 3.2 skizziert grob den Rahmen der Lernphase. Eingabe für diese Phase ist eine Menge an Software-Fehlern sowie deren Korrekturen in Form von Bugfixes. Das Ergebnis der Lernphase sind sogenannte Knotengewichte, die definierte Teile der Software gewichten. Diese dienen später als Eingabe für die Anwendungsphase.

Bedingung 3.1 (Software für die Lernphase)**.**
Die Software für die Lernphase soll die folgenden Bedingungen erfüllen:

1. Die Software enthält Fehler und
2. Es gibt eine Nachfolgerversion der Software, in der die Fehler korrigiert wurden.

Durch die Korrekturen in Punkt 2 der Bedingung 3.1 können die tatsächlichen Ursachen der Fehler automatisch ermittelt werden. Dazu wird ein Vergleich zwischen den beiden Software-Ständen durchgeführt. Aufgrund der Tatsache, dass in dieser Arbeit die Ursachen von Laufzeitfehlern gefunden werden sollen, damit die Fehler korrigiert werden können, gilt an dieser Stelle der Umkehrschluss. Wenn eine Software-Stelle korrigiert wird, die ursprünglich zu einem Laufzeitfehler führte, dann wurde die Ursache des Fehlers korrigiert. Die Ermittlung des Bugfixes ergibt somit die Ursache des eigentlichen Fehlers. Nach der Sammlung von Software-Anwendungen, welche die Bedingungen 3.1 erfüllen, kann die eigentliche Lernphase initiiert werden. Dazu wird die Software in Form eines gerichteten Graphen dargestellt. Die Operatoren und Operanden werden zu den Knoten im Graph, der Signal-/Datenfluss wird durch gerichtete Kanten repräsentiert. Der Typ eines Knotens wird durch den jeweiligen Operator bzw. Operanden beschrieben. Der gefundene Fehler kann nun durch ein Mapping auf einen Knoten projiziert werden, den sogenannten *Fehlerknoten*. Im nächsten Schritt werden alle Pfade ermittelt, die zum jeweiligen Fehlerknoten führen. Dazu können mathematische Verfahren aus der Graphentheorie, wie zum Beispiel die Tiefensuche, verwendet werden. Dabei ist es möglich, dass ein Pfad eine Schleife enthält. Um eine endlose Suche zu unterbinden, stoppt die Suche des aktuellen Pfads, wenn ein Knoten doppelt auftritt. Da aufgrund von Bedingung 3.1(2) die korrigierten Versionen der Software und somit die Ursachen der Fehler vorliegen, können die gefundenen Pfade in zwei disjunkte Mengen unterteilt werden:

- p_U = {Pfade, die **U**rsachen enthalten} und
- p_R = {die **r**estlichen Pfade}.

Damit diese beiden Pfadmengen auch mathematisch unterschieden werden können, sollen sie gewichtet werden. Dabei soll gelten:

Bedingung 3.2.
Die Pfade aus p_U bekommen ein höheres Gewicht zugewiesen als diejenigen aus p_R.

Auf genau diese Unterscheidung in Bedingung 3.2 wird das Verfahren auch in der Anwendungsphase zurückgreifen.

Ein Pfadgewicht wird in Abhängigkeit von den im Pfad enthaltenen Knotentypen und der Pfadlänge, d.h. der Anzahl der enthaltenen Knoten, berechnet:

$$\omega(p) = \frac{\sum\limits_{k \in p} w(k)}{|p|}. \tag{3.1}$$

Hierbei ist p ein Pfad, $k \in p$ sind die Knoten in p, $w(k)$ ist das Gewicht von Knoten k und $|p|$ die Länge des Pfades. Letztere wird verwendet, damit die langen Pfade die kürzeren nicht dominieren. Abgesehen von den Knotengewichten $w(k)$ sind alle anderen Angaben gegeben. Das Ziel der Lernphase ist die Berechnung dieser Knotengewichte, wobei ein Gewicht pro Knotentyp berechnet werden soll.

Um die Bedingung 3.2 mathematisch darstellen zu können, werden zwei Matrizen eingeführt, die auf den Mengen p_U und p_R beruhen:

$$\mathbf{U}(i,j) = \frac{\#\text{Knoten vom Typ } j \text{ in Pfad } i}{\#\text{Knoten in Pfad } i} \ \forall \text{ Pfade } \in p_U \tag{3.2}$$

und analog

$$\mathbf{R}(i,j) = \frac{\#\text{Knoten vom Typ } j \text{ in Pfad } i}{\#\text{Knoten in Pfad } i} \ \forall \text{ Pfade } \in p_R. \tag{3.3}$$

Hier gilt $\mathbf{U} \in \mathbb{R}_+^{m_U \times n}$ und $\mathbf{R} \in \mathbb{R}_+^{m_R \times n}$ wobei m_U und m_R die Anzahl der Pfade in p_U bzw. p_R sind und $n > 0$ die Anzahl der verschiedenen

Knotentypen. Sei nun

$$\mathbf{w} = [w(k_1), \dots, w(k_n)]^\top$$

der Vektor, welcher die Gewichte der Knotentypen k_j, $1 \leq j \leq n$, beschreibt. Für die Berechnung der Gewichte aller Pfade in p_U kann Gleichung (3.1) zu

$$\begin{bmatrix} \omega(p_{U_1}) \\ \omega(p_{U_2}) \\ \vdots \\ \omega(p_{U_{m_U}}) \end{bmatrix} = \mathbf{Uw} \tag{3.4}$$

umgeformt werden. Selbiges gilt auch für die Pfade in p_R:

$$\begin{bmatrix} \omega(p_{R_1}) \\ \omega(p_{R_2}) \\ \vdots \\ \omega(p_{R_{m_R}}) \end{bmatrix} = \mathbf{Rw}. \tag{3.5}$$

Bedingung 3.2 kann jetzt mathematisch formuliert werden. Gesucht ist ein Vektor $\mathbf{w}$, sodass

$$(\mathbf{Uw})_i > (\mathbf{Rw})_j \ \forall\ 1 \leq i \leq m_u, 1 \leq j \leq m_R \tag{3.6}$$

gilt. Diese Ungleichung ist äquivalent zu

$$(\mathbf{Uw})_{\min} > (\mathbf{Rw})_{\max}. \tag{3.7}$$

Dabei ist $(\mathbf{Uw})_{\min}$ der minimale Eintrag des Vektors $\mathbf{Uw}$ und analog $(\mathbf{Rw})_{\max}$ der maximale Eintrag für das Matrix-Vektor-Produkt $\mathbf{Rw}$. Das Ziel ist eine Maximierung der Differenz dieser Min-Max-Werte. Für die

Berechnung von $\mathbf{w}$ wird Gleichung (3.7) deswegen in das nichtlineare Optimierungsproblem

$$\max_{\mathbf{w}\in\mathbb{R}^n} \{(\mathbf{Uw})_{\min} - (\mathbf{Rw})_{\max}\} \tag{3.8}$$

oder äquivalent

$$\min_{\mathbf{w}\in\mathbb{R}^n} \{(\mathbf{Rw})_{\max} - (\mathbf{Uw})_{\min}\} \tag{3.9}$$

umgeformt. Um den Lösungsvektor zu beschränken, wird der Suchraum auf

$$D = \{\mathbf{w} \in \mathbb{R}^n \mid a \leq w_i \leq b \ \forall \ 1 \leq i \leq n\} \subset \mathbb{R}^n \tag{3.10}$$

restringiert mit $a, b \geq 0$ und $a < b$. Das bedeutet, dass jeder Eintrag von $\mathbf{w}$ zwischen a und b liegt. Das resultierende Problem hat somit die Form

$$\min_{\mathbf{w}\in D} \{(\mathbf{Rw})_{\max} - (\mathbf{Uw})_{\min}\}. \tag{3.11}$$

Das Problem (3.11) kann äquivalent beschrieben werden als

$$\min_{\mathbf{w}\in\mathbb{R}^n} \{(\mathbf{Rw})_{\max} - (\mathbf{Uw})_{\min} \mid \mathbf{Sw} \leq \mathbf{o}\}. \tag{3.12}$$

Dabei beschreibt die lineare Restriktion $\mathbf{Sw} \leq \mathbf{o}$ die Ungleichung

$$a \leq w_i \leq b \ \forall \ 1 \leq i \leq n. \tag{3.13}$$

Dazu wird letztere Bedingung zunächst aufgesplittet in zwei Ungleichungen

$$w_i \geq a \Leftrightarrow -w_i \leq -a, i = 1, \ldots, n \tag{3.14}$$

und

$$w_i \leq b, i = 1, \ldots, n. \tag{3.15}$$

Die Ungleichungen (3.14) und (3.15) können nun durch

$$\underbrace{\begin{pmatrix} -1 & & & \\ & -1 & & \\ & & \ddots & \\ & & & -1 \end{pmatrix}}_{=\mathbf{S}_1} \underbrace{\begin{pmatrix} w_1 \\ w_2 \\ \vdots \\ w_n \end{pmatrix}}_{=\mathbf{w}} \leq \underbrace{\begin{pmatrix} -a \\ -a \\ \vdots \\ -a \end{pmatrix}}_{=\mathbf{o}_1} \tag{3.16}$$

und

$$\underbrace{\begin{pmatrix} 1 & & & \\ & 1 & & \\ & & \ddots & \\ & & & 1 \end{pmatrix}}_{=\mathbf{S}_2} \underbrace{\begin{pmatrix} w_1 \\ w_2 \\ \vdots \\ w_n \end{pmatrix}}_{=\mathbf{w}} \leq \underbrace{\begin{pmatrix} b \\ b \\ \vdots \\ b \end{pmatrix}}_{=\mathbf{o}_2} \tag{3.17}$$

in Matrixform dargestellt werden. Bei dieser Darstellung gilt, dass nicht-beschriebene Matrixeinträge den Wert 0 haben. Sei nun

$$\mathbf{S} = \begin{pmatrix} \mathbf{S}_1 \\ \mathbf{S}_2 \end{pmatrix} \tag{3.18}$$

und

$$\mathbf{o} = \begin{pmatrix} \mathbf{o}_1 \\ \mathbf{o}_2 \end{pmatrix}, \tag{3.19}$$

dann beschreibt die Ungleichung $\mathbf{Sw} \leq \mathbf{o}$ genau die Restriktion (3.13).

Durch das Lösen des nichtlinearen Optimierungsproblems (3.11) bzw. (3.12) können die Knotengewichte berechnet werden, welche das Ergebnis der Lernphase ergeben. Zum Lösen von nichtlinearen Optimierungsproblemen gibt es diverse Methoden. Alt (2002) beschreibt eine Vielzahl an Methoden für Probleme unterschiedlichen Aufbaus. Diese können durch die Verfahren

von Geiger und Kanzow (2002) ergänzt werden:

1. Ableitungsfreie Verfahren für unrestringierte Optimierungsprobleme
   a) Nelder und Mead (Alt, 2002, S. 21-26)
   b) Mutations-Selektions-Verfahren (Alt, 2002, S. 31)
2. Ableitungsbasierte Verfahren für unrestringierte Optimierungsprobleme
   a) Newton-Verfahren (Alt, 2002, S. 71-76)
   b) Abstiegsverfahren (Alt, 2002, S. 76-86)
   c) Gradientenverfahren (Alt, 2002, S. 97-99)
   d) Gedämpftes Newton-Verfahren (Alt, 2002, S. 106-117)
   e) Variable-Metrik-/Quasi-Newton-Verfahren (Alt, 2002, S. 121-131)
   f) Verfahren konjugierter Richtungen (Alt, 2002, S. 135-139)
   g) Trust-Region-Verfahren (Alt, 2002, S. 142-155)
3. Ableitungsbasierte Verfahren für restringierte Optimierungsprobleme
   a) Innere-Punkte-Methode (Geiger und Kanzow, 2002, S. 129-187)
   b) Strategie der aktiven Mengen (Geiger und Kanzow, 2002, S. 199-206)
4. Nichtglatte Optimierung
   a) Lagrange Dualität (Geiger und Kanzow, 2002, S. 314-326)
   b) Subgradientenverfahren (Geiger und Kanzow, 2002, S. 366-370)

Um zu prüfen, welche dieser Methoden für das Optimierungsproblem (3.11) bzw. (3.12) anwendbar sind, wird Tabelle 3.1 eingeführt. Aus Darstellungsgründen werden hier Abkürzungen verwendet:

- konv. = konvex,
- ¬leer = nichtleer,
- diff'bar = differenzierbar,
- quadr. = quadratisch,
- $\emptyset$ = keine,
- Gl.-Restr. = Gleichungsrestriktionen,
- Ungl.-Restr. = Ungleichungsrestriktionen und
- nichtlin. = nichtlinear.

Ein „x“ in einer Zelle *(Verfahren,Anforderung)* beschreibt, dass das jeweilige Verfahren die entsprechende Anforderung als notwendig voraussetzt. Ist eine Zelle mit dem Buchstaben „o“ gefüllt, so ist die Anforderung optional für das jeweilige Verfahren und somit nicht zwingend notwendig. Das Symbol „-“ schließt die Voraussetzung für das jeweilige Verfahren aus. Um zu erkennen, welche Verfahren für die Optimierungsprobleme (3.11) und (3.12) infrage kommen, werden die Zellen aus den letzten zwei Zeilen aus Tabelle 3.1 mit denen aus den Zellen für die Verfahren verglichen. Dabei stellt sich heraus, dass das Verfahren 4b), das Subgradientenverfahren, für das Problem (3.11) geeignet ist sowie Verfahren 4a), die Lagrange Dualität, für (3.12). Dieses ist in Tabelle 3.2 verkürzt dargestellt.

In den folgenden Abschnitten werden die beiden Verfahren näher erläutert.

Tabelle 3.1: Anforderungen bzw. Voraussetzungen der erwähnten Methoden. „x“= notwendig, „o“ = optional„„-“ = nicht zutreffend

	Suchraum			Zielfunktion			Restriktionen					
	$\mathbb{R}^n$	$D \subset \mathbb{R}^n$		diff’bar	quadr.	konv.	$\emptyset$	linear		nichtlin.	konv.	diff’bar
		konv.	¬leer					Gl.-Restr.	Ungl.-Restr.			
1a	x	-	-	-	o	o	x	-	-	-	-	-
1b	x	-	-	-	o	o	x	-	-	-	-	-
2a	x	-	-	x	o	o	x	-	-	-	-	-
2b	x	-	-	x	o	o	x	-	-	-	-	-
2c	x	-	-	x	o	o	x	-	-	-	-	-
2d	x	-	-	x	o	o	x	-	-	-	-	-
2e	x	-	-	x	o	o	x	-	-	-	-	-
2f	x	-	-	x	x	o	x	-	-	-	-	-
2g	x	-	-	x	o	o	x	-	-	-	-	-
3a	o	o	o	o	-	o	-	x	x	-	o	o
3b	x	-	-	o	x	o	-	x	x	-	o	o
4a	x	-	-	o	o	o	-	o	o	o	o	o
4b	-	x	x	o	o	x	x	-	-	-	-	-
(3.11)	-	x	x	-	-	x	x	-	-	-	-	-
(3.12)	x	-	-	-	-	x	-	-	x	-	-	-

Tabelle 3.2: Mapping der zwei Optimierungsprobleme auf zwei passende Verfahren. „x“= notwendig, „o“ = optional, „-“ = nicht zutreffend

	Suchraum			Zielfunktion			Restriktionen					
	$\mathbb{R}^n$	$D \subset \mathbb{R}^n$		diff'bar	quadr.	konv.	$\emptyset$	linear		nichtlin.	konv.	diff'bar
		konv.	¬leer					Gl.-Restr.	Ungl.-Restr.			
4a	**x**	-	-	o	o	**o**	-	o	**o**	o	o	o
(3.12)	**x**	-	-	-	-	**x**	-	-	**x**	-	-	-
4b	-	**x**	**x**	o	o	**x**	**x**	-	-	-	-	-
(3.11)	-	**x**	**x**	-	-	**x**	**x**	-	-	-	-	-

3.2.1 Lagrange Dualität

Geiger und Kanzow (2002, Abschnitt 6.2) gehen von dem Optimierungsproblem

$$\min\{f(\mathbf{x}) \mid \mathbf{x} \in X, \mathbf{g}(\mathbf{x}) \leq \mathbf{0}, \mathbf{h}(\mathbf{x}) = \mathbf{0}\} \tag{3.20}$$

aus mit $X \subset \mathbb{R}^n, f : \mathbb{R}^n \to \mathbb{R}, \mathbf{g} : \mathbb{R}^n \to \mathbb{R}^m, \mathbf{h} : \mathbb{R}^n \to \mathbb{R}^p$. Die zugehörige *Lagrange-Funktion* ist definiert als

$$\mathfrak{L}(\mathbf{x}, \boldsymbol{\lambda}, \boldsymbol{\mu}) = f(\mathbf{x}) + \sum_{i=1}^{m} \lambda_i g_i(\mathbf{x}) + \sum_{j=1}^{p} \mu_j h_j(\mathbf{x}) = f(\mathbf{x}) + \boldsymbol{\lambda}^\top \mathbf{g}(\mathbf{x}) + \boldsymbol{\mu}^\top \mathbf{h}(\mathbf{x}) \tag{3.21}$$

mit den Lagrange-Multiplikatoren $\boldsymbol{\lambda} \in \mathbb{R}^m$ und $\boldsymbol{\mu} \in \mathbb{R}^p$ sowie den Restriktionen

$$\mathbf{g}(\mathbf{x}) = (g_1(\mathbf{x}), \ldots, g_m(\mathbf{x}))^\top$$

und

$$\mathbf{h}(\mathbf{x}) = (h_1(\mathbf{x}), \ldots, h_p(\mathbf{x}))^\top.$$

Aus der Lagrange-Funktion kann die *duale Funktion* berechnet werden:

$$q(\boldsymbol{\lambda}, \boldsymbol{\mu}) = \inf_{\mathbf{x} \in X} \mathfrak{L}(\mathbf{x}, \boldsymbol{\lambda}, \boldsymbol{\mu}). \tag{3.22}$$

Daraus ergibt sich das zu (3.20) *duale Problem*:

$$\max\{q(\boldsymbol{\lambda}, \boldsymbol{\mu}) \mid \boldsymbol{\lambda} \geq \mathbf{0}, \boldsymbol{\mu} \in \mathbb{R}^p\}. \tag{3.23}$$

Angewendet auf das Problem (3.12) mit

$$f(\mathbf{w}) = (\mathbf{R}\mathbf{w})_{\max} - (\mathbf{U}\mathbf{w})_{\min},$$

$f : D \to \mathbb{R}$, und

$$\mathbf{g}(\mathbf{w}) = \mathbf{S}\mathbf{w} - \mathbf{o},$$

$\mathbf{g}: D \to \mathbb{R}^{2n}$, mit $\mathbf{S}$ wie in (3.18) und $\mathbf{o}$ wie in (3.19) beschrieben, sieht die Lagrange-Funktion hier wie folgt aus:

$$\mathfrak{L}(\mathbf{w}, \boldsymbol{\lambda}) = f(\mathbf{w}) + \sum_{i=1}^{2n} \lambda_i g_i(\mathbf{w}) = f(\mathbf{w}) + \boldsymbol{\lambda}^\top \mathbf{g}(\mathbf{w}) \tag{3.24}$$

mit $\boldsymbol{\lambda} \in \mathbb{R}^{2n}$. Der Lagrange-Multiplikator $\boldsymbol{\mu}$ wird an der Stelle nicht mehr benötigt, da es keine Funktion $\mathbf{h}$ gibt. Die duale Funktion hat die Form

$$q(\boldsymbol{\lambda}) = \inf_{\mathbf{w} \in D} \mathfrak{L}(\mathbf{w}, \boldsymbol{\lambda}). \tag{3.25}$$

Das zu (3.12) duale Problem wird analog zu (3.23) dargestellt durch

$$\max\{q(\boldsymbol{\lambda}) \mid \boldsymbol{\lambda} \geq \mathbf{0}\} \tag{3.26}$$

und nachfolgend auch mit DP (=Dualproblem) bezeichnet. Diese Begriffe können nun für den folgenden Satz verwendet werden.

Satz 3.1 (Starke Dualität)**.**
$X \subseteq \mathbb{R}^n$ sei nichtleer und konvex. Die beiden Funktionen $f : \mathbb{R}^n \to \mathbb{R}$ und $g_i : \mathbf{R}^n \to \mathbb{R}$ mit $i = 1, \ldots, m$ seien konvex. Zusätzlich gelte, dass $h : \mathbb{R}^n \to \mathbb{R}^p$ affin-linear sei. Seien

$$\inf(P) = \inf\{f(\mathbf{x}) \mid \mathbf{x} \in X, \mathbf{g}(\mathbf{x}) \leq \mathbf{0}, \mathbf{h}(\mathbf{x}) = \mathbf{0}\} \tag{3.27}$$

und

$$\sup(DP) = \sup\{q(\boldsymbol{\lambda}, \boldsymbol{\mu}) \mid \boldsymbol{\lambda} \geq \mathbf{0}, \boldsymbol{\mu} \in \mathbb{R}^p\}. \tag{3.28}$$

Ist $\inf(P)$ endlich und gibt es einen Vektor $\hat{\mathbf{x}}$, der zum relativen Inneren von X gehört und die Eigenschaften

$$g_i(\hat{\mathbf{x}}) < 0, i = 1, \ldots, m$$

und

$$\mathbf{h}(\hat{\mathbf{x}}) = \mathbf{0}$$

besitzt, so ist das duale Problem lösbar und es gilt die Gleichung

$$\sup(DP) = \inf(P). \tag{3.29}$$

Beweis. Verweis auf Geiger und Kanzow (2002, S. 323-326). □

Da in dem hier beschriebenen Problem (3.12) die Restriktionsfunktion $\mathbf{h}$ nicht vorkommt, muss der Begriff *affin-linear* an der Stelle nicht erläutert werden. Für das Problem (3.12) ergeben sich somit folgende Voraussetzungen für die Gültigkeit von Satz 3.29:

1. $D \subset \mathbb{R}^n$ ist nichtleer und konvex,
2. f ist konvex und
3. g_i ist konvex für $i = 1, \dots, 2n$.

Behauptung 3.1. Die zulässige Menge

$$D = \{\mathbf{w} \in \mathbb{R}^n \mid a \leq w_i \leq b \; \forall \; 1 \leq i \leq n\}$$

mit $a, b \geq 0$ und $a < b$ ist nichtleer und konvex.

Beweis. D ist nichtleer, da mindestens die Vektoren

$$\begin{pmatrix} a \\ a \\ \vdots \\ a \end{pmatrix} \in \mathbb{R}^n$$

und

$$\begin{pmatrix} b \\ b \\ \vdots \\ b \end{pmatrix} \in \mathbb{R}^n$$

in D enthalten sind. Zur einfachen Darstellung der Konvexität werden zwei Zwischenschritte durchgeführt:

1. $\underbrace{(1-t)}_{0\leq 1-t\leq 1} \cdot \underbrace{x_i}_{\leq b} + \underbrace{t}_{0\leq t\leq 1} \cdot \underbrace{y_i}_{\leq b} \leq (1-t)\cdot b + t\cdot b = b \in D \;\forall\; 1 \leq i \leq n, 0 \leq t \leq 1$
2. $\underbrace{(1-t)}_{0\leq 1-t\leq 1} \cdot \underbrace{x_i}_{\geq a} + \underbrace{t}_{0\leq t\leq 1} \cdot \underbrace{y_i}_{\geq a} \geq (1-t)\cdot a + t\cdot a = a \in D \;\forall\; 1 \leq i \leq n, 0 \leq t \leq 1$

Aus 1 und 2 zusammen folgt

$$a \leq (1-t)\cdot x_i + t\cdot y_i \leq b$$

für alle $1 \leq i \leq n$, $\mathbf{x}, \mathbf{y} \in D$ und $0 \leq t \leq 1$. Damit gilt $(1-t)\cdot \mathbf{x} + t \cdot \mathbf{y} \in D$ und D ist somit konvex. □

Inhaltlich kann die Situation folgendermaßen beschrieben werden. Eine Menge ist genau dann konvex, wenn jede Verbindungsstrecke zwischen zwei Punkten aus der Menge ebenfalls in der Menge liegt. Für die hier beschriebene Menge gilt:

- $n = 1$: $D = [a, b]$, also ein einfaches Intervall, in dem jede Verbindung zwischen zwei Punkten offensichtlich ebenfalls im Intervall liegt
- $n = 2$: $D = [a, b] \times [a, b]$, also ein Quadrat. Selbst wenn nur Randpunkte der Menge betrachtet werden, so liegt auch hier jede Verbindungsstrecke in dem Quadrat.
- $n = 3$: $D = [a, b] \times [a, b] \times [a, b]$, was einen Würfel beschreibt. Auch dieser erfüllt die genannte Eigenschaft.

Behauptung 3.2. Die verwendete Zielfunktion $f : D \to \mathbb{R}$ mit

$$f(\mathbf{w}) = \max(\mathbf{Rw}) - \min(\mathbf{Uw})$$

ist konvex.

Beweis. Aus Behauptung 3.1 ist bekannt, dass D nichtleer und konvex ist. Damit sind die Grundvoraussetzungen aus Definition 9 erfüllt. Weiterhin gilt:

$$\begin{aligned} f((1-t)\cdot\mathbf{x}+t\cdot\mathbf{y}) &= \max(\mathbf{R}((1-t)\cdot\mathbf{x}+t\cdot\mathbf{y})) - \min(\mathbf{U}((1-t)\cdot\mathbf{x}+t\cdot\mathbf{y})) \\ &= \max((1-t)\cdot\mathbf{Rx}+t\cdot\mathbf{Ry}) - \min((1-t)\cdot\mathbf{Ux}+t\cdot\mathbf{Uy}) \\ &\leq (1-t)\cdot\max(\mathbf{Rx})+t\cdot\max(\mathbf{Ry}) \\ &\quad -(1-t)\cdot\min(\mathbf{Ux})-t\cdot\min(\mathbf{Uy}) \\ &= (1-t)\cdot(\max(\mathbf{Rx})-\min(\mathbf{Ux}))+t\cdot(\max(\mathbf{Ry})-\min(\mathbf{Uy})) \\ &= (1-t)\cdot f(\mathbf{x})+t\cdot f(\mathbf{y}) \end{aligned}$$

für alle $\mathbf{x}, \mathbf{y} \in D$ und $t \in [0, 1]$. Daraus folgt, dass f konvex ist. □

Behauptung 3.3. Gegeben sei die Funktion $\mathbf{g} : \mathbb{R}^n \to \mathbb{R}^{2n}$ mit

$$\mathbf{g}(\mathbf{w}) = \mathbf{Sw} - \mathbf{o}.$$

Die einzelnen Funktionen $g_i : \mathbf{R}^n \to \mathbf{R},\ 1 \leq i \leq 2n$, mit

$$g_i(\mathbf{w}) = \mathbf{S}_{i,1:n}\mathbf{w} - o_i$$

sind konvex. Dabei bezeichnet $\mathbf{S}_{i,1:n}$ die i-te Zeile von $\mathbf{S}$.

Beweis. Es gilt

$$\begin{aligned} g_i((1-t)\cdot\mathbf{x}+t\cdot\mathbf{y}) &= \mathbf{S}_{i,1:n}((1-t)\cdot\mathbf{x}+t\cdot\mathbf{y})-o_i \\ &= (1-t)\cdot\mathbf{S}_{i,1:n}\mathbf{x}+t\cdot\mathbf{S}_{i,1:n}\mathbf{y}-o_i \\ &= (1-t)\cdot\mathbf{S}_{i,1:n}\mathbf{x}-(1-t)\cdot o_i+t\cdot\mathbf{S}_{i,1:n}\mathbf{y}-t\cdot o_i \\ &= (1-t)\cdot(\mathbf{S}_{i,1:n}\mathbf{x}-o_i)+t\cdot(\mathbf{S}_{i,1:n}\mathbf{y}-o_i) \\ &= (1-t)\cdot g_i(\mathbf{x})+t\cdot g_i(\mathbf{y}) \end{aligned}$$

für alle $t \in [0,1]$. Damit gilt

$$g_i((1-t)\cdot\mathbf{x}+t\cdot\mathbf{y}) = (1-t)\cdot g_i(\mathbf{x})+t\cdot g_i(\mathbf{y})$$

für alle $\mathbf{x},\mathbf{y} \in D$ und somit ist g_i konvex für alle $i = 1,\dots,2n$. □

Die letzten Behauptungen haben gezeigt, dass die Voraussetzungen für Satz 3.1 erfüllt sind, sodass dieser hier anwendbar ist. Es ist somit möglich, dass anstelle des nichtglatten Problems (3.12) das vereinfachte duale Problem (3.26) gelöst werden kann.

3.2.2 Subgradientenverfahren

Eine Voraussetzung für den Einsatz des Subgradientenverfahrens ist, dass die zulässige Menge offen und konvex ist. Die Konvexität der Menge D wurde bereits in Behauptung 3.1 bzw. in dem entsprechenden Beweis gezeigt. Jedoch ist D nicht offen, da es sich um ein abgeschlossenes n-dimensionales Intervall handelt. Deswegen wird die zulässige Menge modifiziert, sodass eine offene Menge entsteht:

$$\tilde{D} = \{\mathbf{w} \in \mathbb{R}^n \mid a < w_i < b \;\forall\; 1 \leq i \leq n\} \tag{3.30}$$

mit $a, b \geq 0$ und $a < b$. Im Gegensatz zur Menge D sind hier die Werte a und b von der Menge ausgenommen. Ein Algorithmus zur Lösung

Algorithmus 3.1 Subgradientenverfahren.

1: Wähle $\mathbf{w}^0 \in \tilde{D}$, berechne $m_0 = f(\mathbf{w}^0)$, setze $k = 0$.
2: Falls $\mathbf{w}^k$ einem geeigneten Abbruchkriterium genügt: STOP.
3: Berechne $\mathbf{0} \neq \mathbf{s}^k \in \partial f(\mathbf{w}^k)$, setze $\mathbf{d}^k = -\mathbf{s}^k/||\mathbf{s}^k||$ und

$$\mathbf{w}^{k+1} = \mathrm{Proj}_{\tilde{D}}[\mathbf{w}^k + t_k \mathbf{d}^k]$$

für eine Schrittweite $t_k > 0$.
4: Berechne $m_{k+1} = \min\{f(\mathbf{w}^{k+1}), m_k\}$.
5: Setze $k \leftarrow k + 1$ und gehe zu 2.

des Problems (3.11) ist in dem von Geiger und Kanzow (2002, S. 367) übernommenen Algorithmus 3.1 dargestellt. Eine mögliche Wahl der Schrittweite t_k ist $t_k = \dfrac{1}{k+1}$.

Satz 3.2 stellt klar, dass dieses Verfahren für das gegebene Problem konvergiert.

Satz 3.2. Das Optimierungsproblem

$$\min_{\mathbf{w} \in \tilde{D}} \{f(\mathbf{w})\} \tag{3.31}$$

mit

$$f(\mathbf{w}) = (\mathbf{R}\mathbf{w})_{\max} - (\mathbf{U}\mathbf{w})_{\min}$$

besitze eine nichtleere Lösungsmenge und

$$f^* = \min\{f(\mathbf{w}) \mid \mathbf{w} \in \tilde{D}\}$$

bezeichne den optimalen Funktionswert. Seien $\{\mathbf{w}^k\}$ und $\{m_k\}$ die von Algorithmus 3.1 erzeugten Folgen, wobei die Schrittweiten $t_k \downarrow 0$ und $\sum\limits_{k=0}^{\infty} t_k = +\infty$ genügen mögen. Dann konvergiert die Folge $\{m_k\}$ gegen f^*.

Beweis. Verweis auf Geiger und Kanzow (2002, S. 367-368). □

Für die oben genannte mögliche Schrittweite t_k gilt offensichtlich $t_k \downarrow 0$. Die harmonische Reihe $\sum_{k=0}^{\infty} \frac{1}{k+1} = \sum_{k=1}^{\infty} \frac{1}{k}$ divergiert (vgl. Hildebrandt, 2002, S. 64), sodass die Bedingungen für die Schrittweite aus Satz 3.2 erfüllt sind.

3.2.3 Beispiel Lernphase

Gegeben seien zwei Fehler, deren jeweilige Graphenstruktur in den Abbildungen 3.3 und 3.4 dargestellt ist. Die Buchstaben $a, b, c, \ldots, i$ beschreiben neun verschiedene Knotentypen, wobei der Fehlerknoten jeweils vom Typ a ist. Anhand der Graphenstrukturen lassen sich insgesamt fünf Pfade ableiten:

$$
\begin{aligned}
&p_1 : a \to b \to c \to d, \\
&p_2 : a \to e \to f \to g \to b \to b \to h \to b \to i \to d, \\
&p_3 : a \to e \to f \to g \to a \text{ (Schleife) }, \\
&p_4 : a \to e \to f \to g \to e \to g \to b \to b \to h \to b \to i \to d \text{ und} \\
&p_5 : a \to h \to b \to c \to i \to c \to d,
\end{aligned}
$$

wobei p_1, p_2 und p_3 von Fehler 1 und p_4 sowie p_5 von Fehler 2 stammen. Aufgrund der Korrekturen in den nachfolgenden Versionen ist bekannt, dass die Ursache von Fehler 1 in p_1 und die Ursache von Fehler 2 in p_5 enthalten ist. Der Knotengewichtevektor $\mathbf{w}$ beschreibt hier die Gewichte der Knotentypen $a, \ldots, i$:

$$
\mathbf{w} = \begin{pmatrix} w(a) & w(b) & w(c) & w(d) & w(e) & w(f) & w(g) & w(h) & w(i) \end{pmatrix}^\top . \tag{3.32}
$$

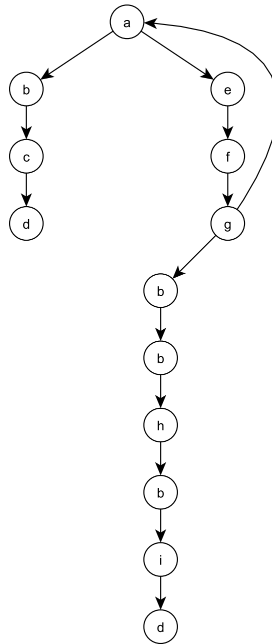

Abbildung 3.3: Graphenstruktur von Fehler 1 in Abschnitt 3.2.3.

Die Reihenfolge dieses Vektors ist bei der Erstellung der Matrizen **U** und **R** zu beachten:

$$\mathbf{U} = \begin{array}{c} \\ p_1 \\ p_5 \end{array} \begin{array}{c} \begin{array}{ccccccccc} a & b & c & d & e & f & g & h & i \end{array} \\ \begin{pmatrix} 1/4 & 1/4 & 1/4 & 1/4 & 0 & 0 & 0 & 0 & 0 \\ 1/7 & 1/7 & 2/7 & 1/7 & 0 & 0 & 0 & 1/7 & 1/7 \end{pmatrix} \end{array} \quad (3.33)$$

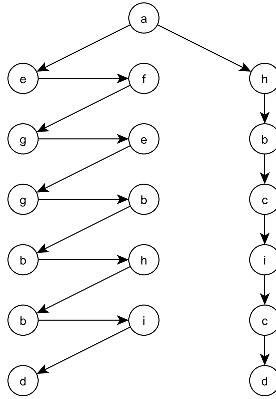

Abbildung 3.4: Graphenstruktur von Fehler 2 in Abschnitt 3.2.3.

und

$$\mathbf{R} = \begin{array}{c} \\ p_2 \\ p_3 \\ p_4 \end{array} \begin{array}{c} \begin{array}{ccccccccc} a & b & c & d & e & f & g & h & i \end{array} \\ \begin{pmatrix} 1/10 & 3/10 & 0 & 1/10 & 1/10 & 1/10 & 1/10 & 1/10 & 1/10 \\ 2/5 & 0 & 0 & 0 & 1/5 & 1/5 & 1/5 & 0 & 0 \\ 1/12 & 3/12 & 0 & 1/12 & 2/12 & 1/12 & 2/12 & 1/12 & 1/12 \end{pmatrix} \end{array}. \tag{3.34}$$

Ein lokales Minimum des nichtlinearen Optimierungsproblems

$$\min_{\mathbf{w} \in D} f(\mathbf{w}) \tag{3.35}$$

mit

$$f(\mathbf{w}) = (\mathbf{Rw})_{\max} - (\mathbf{Uw})_{\min} \tag{3.36}$$

und

$$D = \{\mathbf{w} \in \mathbb{R}^9 \mid 1 \leq w_i \leq 100 \; \forall \; i = 1, \ldots, 9\}$$

ist der Gewichtevektor

$$\mathbf{w_1} = \begin{pmatrix} 98,27 & 19,76 & 100 & 100 & 1 & 1 & 1 & 69,26 & 69,26 \end{pmatrix}^\top . \tag{3.37}$$

Mit diesen Werten gilt

$$f^* = (\mathbf{Uw})_{\min} - (\mathbf{Rw})_{\max} = 39,6, \tag{3.38}$$

sodass Gleichung (3.7) erfüllt ist. Wird stattdessen das Problem

$$\min_{\mathbf{w}\in\mathbb{R}^9} \{f(\mathbf{w}) \mid \mathbf{Sw} \leq \mathbf{o}) \tag{3.39}$$

verwendet mit

$$\mathbf{S} = \begin{pmatrix} -1 & & & \\ & -1 & & \\ & & \ddots & \\ & & & -1 \\ 1 & & & \\ & 1 & & \\ & & \ddots & \\ & & & 1 \end{pmatrix} \in \mathbb{R}^{18\times 9} \tag{3.40}$$

und

$$\mathbf{o} = \begin{pmatrix} -1 \\ -1 \\ \vdots \\ -1 \\ 100 \\ 100 \\ \vdots \\ 100 \end{pmatrix} \in \mathbb{R}^{18}, \tag{3.41}$$

so ist

$$\mathbf{w}_2 = \begin{pmatrix} 67,08 & 1,05 & 100 & 100 & 1 & 1 & 1 & 50,77 & 50,33 \end{pmatrix}^\top \tag{3.42}$$

ein lokales Minimum, welches den gleichen Optimalwert wie in (3.38) erzeugt. Daran ist erkennbar, dass die Lösung nicht eindeutig ist. Aus diesem Grund muss vor Anwendung des Verfahrens geklärt werden, welche Form des Optimierungsproblems gelöst werden soll. Dieses wird im Rahmen der Anwendung im Kapitel 4.2.3 diskutiert.

3.3 Anwendungsphase in der Theorie

Im Gegensatz zur Lernphase, die einmal für die vorgegebene Lernmenge ausgeführt wird, wird die Anwendungsphase für jeden Fehler separat ausgeführt. Wie schon in Abschnitt 3.2 erklärt, dienen die in der Lernphase berechneten Knotengewichte als Eingangsdaten für die Anwendungsphase. Zusätzlich dazu gibt es einen gegebenen Fehler, der durch Testen, statische Analyse oder weitere Analysemöglichkeiten identifiziert werden kann. Das Ergebnis der Anwendungsphase ist ein Ranking aller Pfade, die zu diesem Fehler führen (vgl. Abbildung 3.5). Je weiter vorne ein Pfad im Ranking steht, desto höher soll die Wahrscheinlichkeit sein, dass dieser Pfad die

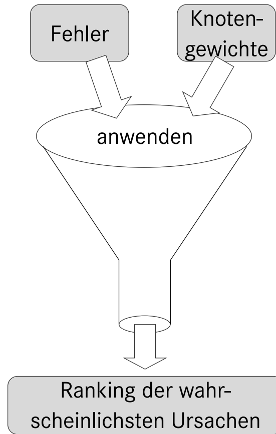

Abbildung 3.5: Anwendungsphase.

Ursache enthält. Die ersten Schritte der Anwendungsphase sind ähnlich zu denen der Lernphase. Zunächst wird die Software wieder als gerichteter Graph betrachtet und das Mapping von dem Fehler auf den Fehlerknoten ausgeführt. Auch hier werden alle Pfade ermittelt, die zu diesem Knoten führen. Im Gegensatz zur Lernphase gibt es in der Anwendungsphase nur eine Art von Pfaden, da hier noch nicht bekannt ist, welche der Pfade die Ursachen enthalten. Sei also p_A die Menge aller gefundenen Pfade. Daraus wird analog wie in Abschnitt 3.2 eine Matrix $\mathbf{A} \in \mathbb{R}_+^{m \times n}$ eingeführt, welche die Menge p_A mit m Pfaden mathematisch beschreibt:

$$\mathbf{A}(i, j) = \frac{\#\text{Knoten vom Typ } j \text{ in Pfad } i}{\#\text{Knoten in Pfad } i} \; \forall \text{ Pfade } \in p_A. \tag{3.43}$$

Die Anzahl der Spalten n entspricht der Anzahl der zuvor berechneten Knotengewichte. Dabei ist zu beachten, dass die Reihenfolge der Spalten von $\mathbf{A}$ genau der des Gewichtevektors $\mathbf{w}$ entspricht. Sollten an der Stelle Knoten auftreten, die in der Lernphase nicht gewichtet wurden, so wird

deren Gewicht durch den Wert 0 beschrieben. Der aktuelle Fehler kann in einer weiteren Iteration an die Lernphase übergeben werden, damit die verbleibenden Knotentypen ebenfalls gewichtet werden. Die Multiplikation von $\mathbf{A}$ mit dem Knotengewichtevektor $\mathbf{w}$ aus der Lernphase ergibt den Pfadgewichtevektor $\boldsymbol{\omega}$:

$$\boldsymbol{\omega} = \begin{pmatrix} \omega(p_1) \\ \omega(p_2) \\ \vdots \\ \omega(p_m) \end{pmatrix} = \mathbf{A}\mathbf{w}. \tag{3.44}$$

Die Einträge von $\boldsymbol{\omega}$ werden nun nach absteigendem Gewicht sortiert. Dazu wird der Vektor $\overline{\boldsymbol{\omega}}$ eingeführt,

$$\overline{\boldsymbol{\omega}} = \begin{pmatrix} \omega(p_{i_1}) \\ \omega(p_{i_2}) \\ \vdots \\ \omega(p_{i_m}) \end{pmatrix}, \tag{3.45}$$

mit

$$\omega(p_{i_1}) \geq \omega(p_{i_2}) \geq \ldots \geq \omega(p_{i_m})$$

und $\{i_1, i_2, \ldots, i_m\}$ eine Permutation von $\{1, 2, \ldots, m\}$. Jetzt gilt: je weiter oben ein Pfad p_i durch die Sortierung in $\overline{\boldsymbol{\omega}}$ verschoben wurde, desto höher soll die Wahrscheinlichkeit sein, dass p_i die Ursache enthält. Die Sortierung beschreibt somit das initial erwähnte Ranking der Pfade.

3.3.1 Beispiel Anwendungsphase

Abbildung 3.6 zeigt die Graphenstruktur gegebener Software. Für die Operation, die im Graphen durch den Knoten h repräsentiert wird, wurde ein

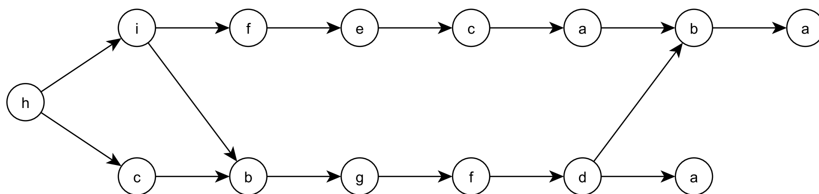

Abbildung 3.6: Graphenstruktur für das Beispiel der Anwendungsphase in Abschnitt 3.3.1.

Fehler identifiziert. Zu diesem Fehler führen fünf Pfade:

$$\begin{aligned}
p_1 &: h \to i \to f \to e \to c \to a \to b \to a, \\
p_2 &: h \to c \to b \to g \to f \to d \to b \to a, \\
p_3 &: h \to i \to b \to g \to f \to d \to b \to a, \\
p_4 &: h \to c \to b \to g \to f \to d \to a \text{ und} \\
p_5 &: h \to i \to b \to g \to f \to d \to a.
\end{aligned}$$

Die daraus resultierende Matrix ist

$$\mathbf{A} = \begin{array}{c} \\ p_1 \\ p_2 \\ p_3 \\ p_4 \\ p_5 \end{array} \begin{array}{c} \begin{array}{ccccccccc} a & b & c & d & e & f & g & h & i \end{array} \\ \begin{pmatrix} 2/8 & 1/8 & 1/8 & 0 & 1/8 & 1/8 & 0 & 1/8 & 1/8 \\ 1/8 & 2/8 & 1/8 & 1/8 & 0 & 1/8 & 1/8 & 1/8 & 0 \\ 1/8 & 2/8 & 0 & 1/8 & 0 & 1/8 & 1/8 & 1/8 & 1/8 \\ 1/7 & 1/7 & 1/7 & 1/7 & 0 & 1/7 & 1/7 & 1/7 & 0 \\ 1/7 & 1/7 & 0 & 1/7 & 0 & 1/7 & 1/7 & 1/7 & 1/7 \end{pmatrix} \end{array}. \qquad (3.46)$$

Diese Pfade werden nun mittels der Knotengewichte aus (3.37) gewichtet. Daraus ergeben sich folgende Pfadgewichte:

$$\boldsymbol{\omega}^1 = \begin{pmatrix} \omega^1(p_1) \\ \omega^1(p_2) \\ \vdots \\ \omega^1(p_5) \end{pmatrix} = \mathbf{A}\mathbf{w_1} = \begin{pmatrix} 57,1 \\ 51,13 \\ 47,29 \\ 55,61 \\ 51,22 \end{pmatrix}. \tag{3.47}$$

Das Sortieren nach absteigendem Pfadgewicht ergibt die Priorisierung

$$p_1 \to p_4 \to p_5 \to p_2 \to p_3.$$

Anstatt beim Review in einer beliebigen Reihenfolge der Pfade vorzugehen, kann die obige Reihenfolge verwendet werden, um die Ursache zu finden. Die Verwendung des zweiten Lösungsvektors in (3.42) ergibt

$$\boldsymbol{\omega}^2 = \begin{pmatrix} \omega^2(p_1) \\ \omega^2(p_2) \\ \vdots \\ \omega^2(p_5) \end{pmatrix} = \mathbf{A}\mathbf{w_2} = \begin{pmatrix} 42,29 \\ 40,24 \\ 34,04 \\ 45,84 \\ 38,75 \end{pmatrix}. \tag{3.48}$$

Es zeigt sich, dass die Priorisierung von der zuvor genannten abweicht:

$$p_4 \to p_1 \to p_2 \to p_5 \to p_3.$$

Im nächsten Kapitel wird gezeigt, durch welches Werkzeug das nichtlineare Optimierungsproblem gelöst wird. Aus Gründen der Einfachheit wird dabei die Form

$$\min_{\mathbf{w}\in\mathbb{R}^n} \{f(\mathbf{w}) \mid \mathbf{S}\mathbf{w} \leq \mathbf{o}\}$$

verwendet, da die Ungleichungsrestriktion $\mathbf{Sw} \leq \mathbf{o}$ leichter darzustellen ist als die Einschränkung des Suchraums.

4 Verfahren in der Praxis

Die in dieser Arbeit beschriebene Methode wird nachfolgend auf Software angewendet, die modellbasiert entwickelt wurde. Dafür werden Modelle der Daimler AG aus unterschiedlichen Bereichen des PKW-Sektors verwendet, die sich derzeit im Entwicklungszustand befinden. Beispielanwendungen sind Lichtsteuergeräte, Steuergeräte für die Sitzheizung oder die Steuerung der Seiten- oder Dachfenster. Die zugrunde liegenden Modelle sind in Simulink von der Firma MathWorks und TargetLink von der Firma dSPACE entwickelt worden.

4.1 Modellbasierter Entwicklungsprozess

> „Durch die Einführung der modellbasierten Entwicklung wurde es dem Entwickler ermöglicht, Systeme zu modellieren anstatt diese programmieren zu müssen. Modellierungssprachen bieten ausdrucksstärkere Modellierungselemente und abstrahieren weitgehend von der Implementierung. Anstatt mit Variablen, Schleifen und Funktionsaufrufen zu arbeiten, stehen in Modellierungssprachen Kon-

> strukte wie Übertragungsfunktionen, Fouriertransformationen oder Zustandsautomaten zur Verfügung. Der eigentliche Quellcode wird dann nicht mehr manuell geschrieben, sondern weitgehend automatisch generiert. Die Modelle stellen somit nicht mehr eine inkonsistente Dokumentation dar, sondern sind das zentrale Entwicklungsartefakt der Entwickler.“ (Berns et al., 2010)

Die nachfolgende Abschnitte detaillieren den Entwicklungsprozess aus Abbildung 1.6.

4.1.1 Anforderungen

Der erste Prozessschritt in der modellbasierten Entwicklung ist die Erstellung eines sogenannten Lastenhefts zur Dokumentation der Anforderungen. Diese Anforderungen können sich einerseits auf die Funktionalität der Software, andererseits aber ebenfalls auf das Layout des Modells beziehen. Hier gibt es beispielsweise Anforderungen an die Farbe von Blöcken, ob sich Signale kreuzen dürfen, wie Signale zu orientieren sind, usw. Des Weiteren kann gefordert werden, dass Regeln wie zum Beispiel die der MISRA (Motor Industry Software Reliability Association) (Mira Ltd, 2013) oder der ISO26262 (International Organization for Standardization (ISO), 2012) eingehalten werden müssen. Anhand dieser Anforderungen können die Modelle entwickelt werden.

4.1.2 Code-Generierung

Ausgehend von einem Modell kann C-Code generiert werden, der dem inhaltlichen Verhalten des Modells entspricht. Dafür wird in der vorliegenden Arbeit TargetLink verwendet. Vorbereitet für die Code-Generierung in TargetLink sieht das Modell aus Abbildung 1.5 ein wenig modifiziert aus, siehe Abbildung 4.1. Im Rahmen der Vorbereitung wird ein Subsys-

tem eingefügt, für welches die Code-Generierung durchgeführt wird. Der generierte Code ist in Code 4.1 gegeben.

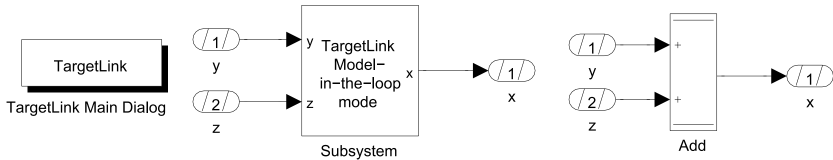

Abbildung 4.1: Modell aus Abbildung 1.5 für die Code-Generierung durch TargetLink vorbereitet.

```
Int16 x;
Int16 y;
Int16 z;


Void Subsystem(Void)
{
   /* TargetLink outport: Subsystem/x
      # combined # Sum: Subsystem/Add */
   x = (Int16) (y + z);
}
```

Code 4.1: Durch TargetLink generierter C-Code für das Modell in Abbildung 4.1.

4.1.3 Code-Analyse

Im Anschluss erfolgt eine Analyse des generierten Codes. Das für diese Arbeit verwendete Werkzeug dafür ist Polyspace der Firma MathWorks (2014d). Polyspace verwendet die Methode der statischen Analyse, um

zum Beispiel C- oder C++-Code zu analysieren. Dabei wird der Code bzgl. der folgenden Laufzeitfehler analysiert:

- *„Überlauf, Unterlauf, Division durch Null und andere arithmetische Fehler,*
- *unzulässiger Array-Zugriff und unerlaubt dereferenzierte Zeiger,*
- *immer wahr/falsch-Anweisung,*
- *nicht-initialisierte Klassenmember (C++),*
- *Lesezugriff auf nicht initialisierte Daten,*
- *Zugriff, um [...] Zeiger (C++) für ungültig zu erklären,*
- *toter Code,*
- *dynamische Fehler in Bezug auf Objektprogrammierung, Vererbung und Ausnahmebehandlung (C++)“* (vgl. MathWorks, 2014b).

Das Ergebnis der statischen Analyse wird in Polyspace anhand einer farbigen Markierung des Codes dargestellt:

- grün: es ist bewiesen, dass an dieser Stelle keiner der untersuchten Laufzeitfehler auftreten kann,
- rot: an der Stelle tritt definitiv ein Laufzeitfehler auf,
- orange: an der Stelle kann möglicherweise ein Laufzeitfehler auftreten,
- grau: diese Stelle ist nicht erreichbar (*Unreachable/Dead Code*).

Ein Grund für eine orange Meldung ist zum Beispiel eine Überapproximation des Werkzeugs, die durch das verwendete Analyseverfahren oder fehlende Analyseangaben des Entwicklers zustande kommen kann.

Statische Analyse

Eine Analyse wird als *statisch* bezeichnet, wenn die Software während des Analyseprozesses nicht ausgeführt wird. Das Gegenteil der statischen Analyse ist die *dynamische Analyse*. Ein Beispiel für eine dynamische Analyse ist das *Testen*. Hier werden der Software zuvor definierte Eingaben übergeben, der sogenannte *Testfall*, sodass die Software mit diesen Eingaben ausgeführt wird. Polyspace verwendet das statische Verfahren der *abstrakten Interpretation*.

Abstrakte Interpretation

Das Prinzip der *abstrakten Interpretation* beruht auf der Abstraktion von Werten. Anstatt jeden einzelnen Wert zu analysieren, erfolgt hier eine Analyse der Abstraktion. Angenommen, es sei die Funktion

$$f(x,y) = \frac{1}{x-y} \tag{4.1}$$

gegeben. Ein möglicher Laufzeitfehler an dieser Stelle ist eine Division durch Null, wodurch ein nicht definiertes Verhalten des Programms auftreten kann. Die Division durch Null geschieht genau dann, wenn x und y identisch sind. Deswegen soll geprüft werden, ob der Fall $x = y$ eintreten kann. Für die Überprüfung sei eine gewisse Anzahl an Paaren (x, y) aus einer definierten Menge M gegeben. Als Behauptung gilt:

Behauptung 4.1. $x \neq y \; \forall \; (x, y) \in M.$

Bei der Darstellung dieses Beispiels in den Abbildungen 4.2 und 4.3 wurde die Idee der Darstellung von ELEKTRONIKPRAXIS (2010) übernommen. In Abbildung 4.2 sind die Paare (x, y) sowie die Funktion $y = x$ vorgegeben. Bei der dynamischen Methode des Testens würden hier endlich viele Kombinationen der x- und y-Werte getestet werden. Da die Anzahl der Testfälle mit der Anzahl der gegebenen Punkte steigt, steigt ebenfalls der

Testaufwand. Um diesen Aufwand zu reduzieren, werden Abstraktionen gebildet. Diese sind in Abbildung 4.3 durch die Polyeder dargestellt. Anstelle jeden einzelnen Punkt zu prüfen, ist es ausreichend, wenn die Polyeder auf Schnittstellen mit der Gleichung $y = x$ geprüft wird. Keine der Abstraktionen schneidet die Funktion, deswegen kann der Fall $x = y$ nicht eintreten. Behauptung 4.1 ist somit bewiesen.

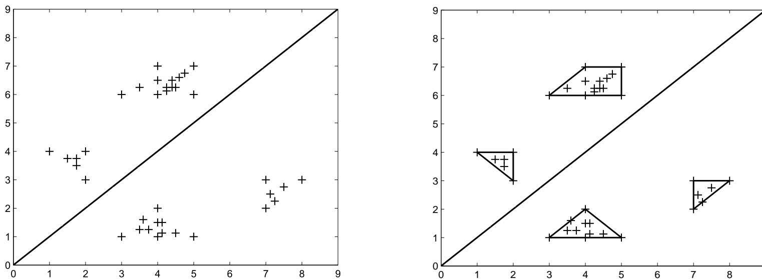

Abbildung 4.2: Gleichung $y = x$ sowie zu prüfende Punkte.

Abbildung 4.3: Abstraktion bilden.

In Code 4.2 ist ein Beispiel dargestellt, welches durch Polyspace analysiert wird. Die hier zu prüfende Behauptung ist:

Behauptung 4.2. Der Code 4.2 enthält keinen Overflow.

```
1  #include <stdio.h>
2
3  int main(){
4          char x;
5          x = 1;
6
7          while (x<128){
8                  x = x + 1;
9          }
10         return 0;
11 }
```

Code 4.2: C-Code, der einen Overflow enthält.

In Abbildung 4.4 werden Informationen zu der Variable `x` in Zeile `5` angefragt. Hier wird durch Polyspace bestätigt, dass die Zuweisung des Wertes `1` der Variablen `x` innerhalb des spezifizierten Datentyps, hier `char`, liegt.

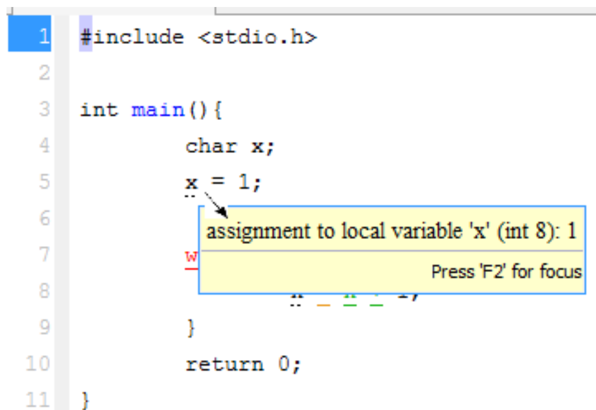

Abbildung 4.4: Wertzuweisung Zeile `5`.

In der `while`-Schleife in Zeile `7` prüft Polyspace, ob der dadurch entstandene Wertebereich noch immer für den angegebenen Datentypen gültig

ist. Durch die Bedingung x<128 entsteht der Wertebereich $]-\infty, 127]$. Aus Zeile 5 geht jedoch bereits hervor, dass x mindestens den Wert 1 annehmen muss. Deswegen entsteht an dieser Stelle das Intervall $[1, 127]$, siehe Abbildung 4.5. Das mögliche Intervall für den Wertebereich von

```
1   #include <stdio.h>
2
3   int main(){
4           char x;
5           x = 1;
6
7           while (x<128){
8
9           }
10          return 0
11  }
```

local variable 'x' (int 8): [1 .. 127]
conversion from int 8 to int 32
right: [1 .. 127]
result: [1 .. 127]
Press 'F2' for focus

Abbildung 4.5: Wertebereichsbestimmung in while-Bedingung.

x bleibt in Zeile 8 auf der rechten Seite der Zuweisung identisch mit dem der while-Schleife, wie es in Abbildung 4.6 gezeigt wird. Auf das mögliche Intervall wird nun der Wert 1 addiert. In Polyspace bedeutet diese Operation [1,127] + [1,1] = [2,128]. An der Stelle wird der neu berechnete Wert noch nicht zugewiesen, deswegen wird zu dem Zeitpunkt keine Meldung durch Polyspace ausgegeben. Diese Zuweisung erfolgt im nächsten Schritt. Der Variable x wird das Intervall [2,128] zugewiesen, welches außerhalb des definierten Datentypbereichs liegt, siehe Abbildung 4.8. Die Polyspace-Meldung hier ist „operation [conversion from int32 to int8] on scalar may overflow (results strictly greater than MAX INT8)“. Die while-Schleife ist markiert durch die Meldung „NTL - Non Terminating Loop“, also als Endlosschleife. Eine mögliche Fehlerkorrektur ist die Änderung der Bedingung in der While-Schleife zu „while (x<127)“. In diesem Beispiel wird nur auf den Fehlertyp Over-

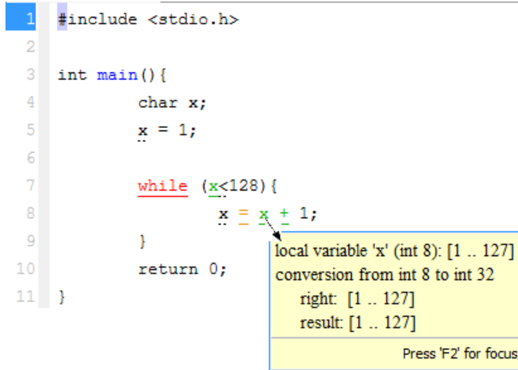

Abbildung 4.6: Polyspace-Analyse: Wertebereichsbestimmung für `x` auf der rechten Seite der Zuweisung.

flow eingegangen.

In der vorliegenden Dissertation werden Ergebnisse von Polyspace-Analysen verwendet, wie sie von einem Entwickler durchgeführt werden. Dieser ist für gewöhnlich kein Experte für diese Art von Analysen. Aus dem Grund wird bei der Bewertung von orangefarbenen Meldungen ein Model-Checker zur Unterstützung verwendet. Diese Kombination aus abstrakter Interpretation und Model-Checking ist bereits bekannt, wie zum Beispiel durch Post et al. (2008). Dabei wird die von Polyspace ausgegebene Meldung in eine mathematische Beschreibung übersetzt. Der Overflow in Code 4.2 kann zum Beispiel durch `x > 127` durch den Model-Checker bestätigt werden. Die mathematische Beschreibung ist somit abhängig von der Art der Warnung von Polyspace sowie von dem jeweiligen Code-Abschnitt.

4.1.4 Ursachenidentifikation

Nachdem im generierten Code Fehler identifiziert wurden, müssen diese korrigiert werden. Für diesen Schritt ist es zwingend notwendig, dass die

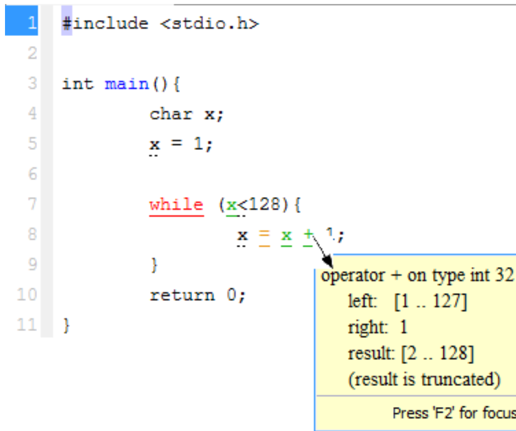

Abbildung 4.7: Polyspace-Analyse: Wertebereichsbestimmung der Addition.

Ursache, d.h. der Auslöser des Fehlers, bekannt ist. Das theoretische Verfahren zur Ursachenidentifikation wurde bereits in Kapitel 3 dargestellt. Für weitere Informationen zur Implementierung des Verfahrens wird an dieser Stelle auf den Abschnitt 4.2 verwiesen.

4.1.5 Korrektur von Fehlern

Nach der Lokalisierung der Ursachen ist der nächste Schritt die Korrektur der Fehler. Dafür gibt es bei der modellbasierten Entwicklung zwei Möglichkeiten:

1. die Korrektur des Fehlers erfolgt im generierten Code oder
2. die Korrektur des Fehlers erfolgt im Modell selbst.

Bei Möglichkeit 1 findet die Korrektur lokal im Code statt. Wird der Code nun auf das Steuergerät geladen, so ist der zu behebende Fehler nicht

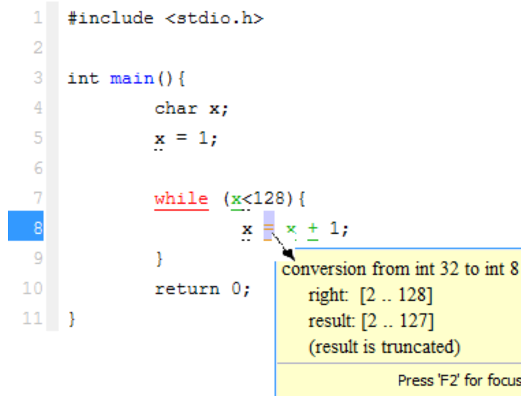

Abbildung 4.8: Polyspace-Analyse: bei der Zuweisung entspricht der Wertebereich nicht dem angegebenen Datentypen.

mehr vorhanden. Diese Vorgehensweise ist unter anderem genau dann sinnvoll, wenn sichergestellt ist, dass der Code nur einmalig generiert wird. Bei der modellbasierten Entwicklung werden Modelle jedoch häufig weiterentwickelt oder Teile davon für andere Modelle verwendet, sodass die Code-Generierung wiederholt wird. In den neuen Modellen und dem neu generierten Code wäre der Fehler somit weiterhin enthalten. Deswegen ist es sinnvoll, dass alle gefundenen Fehler im Modell selbst korrigiert werden.

4.2 Implementierung

Für alle bisher beschriebenen Methoden wird nachfolgend erläutert, wie sie implementiert werden. Hierzu werden die verwendeten Werkzeuge, Sprachen und Methoden beschrieben.

4.2.1 Übersicht über implementierte Funktionen

Alle für diese Dissertation geschriebenen Skripte sind in Matlab in der Version R2011b implementiert. Matlab ist eine Sprache, die für numerische Berechnungen, Datenanalyse und -visualisierung, Programmierung und Algorithmenentwicklung sowie Anwendungsentwicklung und -bereitstellung verwendet werden kann, siehe MathWorks (2014a). Grund für die Wahl von Matlab für die Implementierung ist, dass Matlab die Basis für die verwendeten Tools Simulink (Version R2011b), TargetLink (Version 3.4), Model Compare (Version 2.5), Polyspace (Version R2012b) und Embedded Validator (Version 4.2) ist, sodass aus Matlab heraus sämtliche genannten Werkzeugen aufrufbar sind. Die Schnittstellen der implementierten Skripte sind in Abbildung 4.9 dargestellt. Das Skript `Schreibe_Pfade.m`, welches die Funktion `DFS_Visit.m` aufruft, welche wiederum auf `Adj.m` zugreift, ermittelt alle Pfade, die zu einem spezifizierten Block führen. Der dahinterstehende Algorithmus ist im Abschnitt „Pfadsuche“ in Kapitel 4.2.3 angegeben. Die Pfade werden anhand ihrer Blocktypen in einer Textdatei gespeichert. Die Berechnung der Blockgewichte erfolgt in dem Skript `Berechne_Blockgewichte`. Dabei wird auf `Zielfunktion.m` und `Matrizen_fuer_Gewichtung.m` zugegriffen. Erstere Funktion beschreibt die Zielfunktion für die nichtlineare Optimierung. In der zweiten Funktion werden die Matrizen $\mathbf{U}$ und $\mathbf{R}$ anhand der ermittelten Pfade berechnet, wie es bereits in Kapitel 3.2 beschrieben wurde. Analog dazu berechnet `Matrix_fuer_Evaluierung.m` die Matrix $\mathbf{A}$, die alle Pfade zum spezifizierten Block in der Anwendungsphase beschreibt (vgl. Kapitel 3.3). Diese berechnete Matrix wird in der Funktion `Anwendungsphase.m` verwendet, um die Pfade mittels der in der Lernphase berechneten Blockgewichte zu gewichten. Nach der Gewichtung der Pfade werden sie nach absteigendem Gewicht sortiert. Diejenigen Pfade mit höchstem Gewicht werden in der Dokumentation der Pfade markiert.

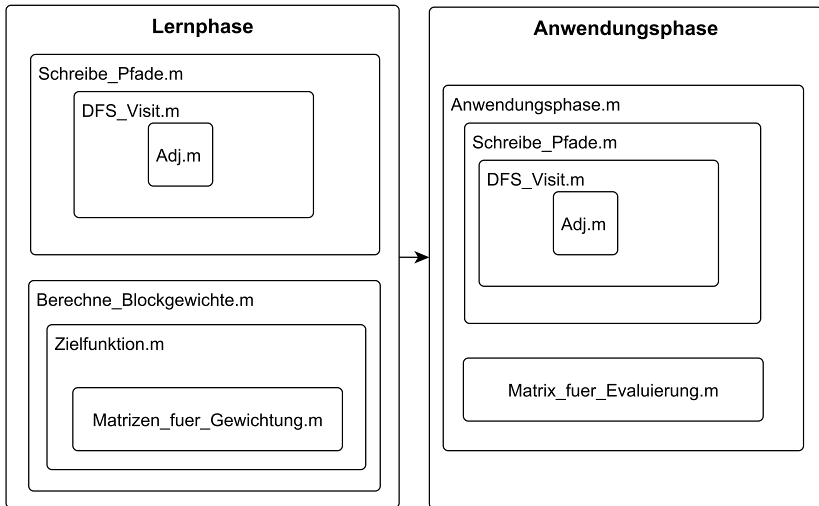

Abbildung 4.9: Übersicht über geschriebene Matlab-Skripte.

4.2.2 Vom Modell zum Graphen

In Abbildung 4.10 ist ein Auszug der Blockmenge von Simulink dargestellt. Jeder Block wird durch einen Blocktypen beschrieben. Der Block unten links in Abbildung 4.10 ist zum Beispiel vom Typ *Relational Operator* und hat zur Aufgabe, die eingehenden Signale miteinander zu vergleichen. Der in Kapitel 3 verwendete Begriff *Knotentyp* wird an der Stelle durch *Blocktyp* ersetzt und nachfolgend nur noch auf den Typen eines Simulink-/TargetLink-Blocks bezogen. Da das Verfahren, wie es in Kapitel 3 beschrieben ist, auf Graphen basiert, muss das Modell in einen Graphen transformiert werden. Dazu werden die Blöcke im Modell als Knoten im Graphen angenommen. Die Kanten zwischen den Knoten entstehen durch den Signal- bzw. Datenfluss im Modell. Von diesem werden auch die Richtungen der Kanten abgelesen.

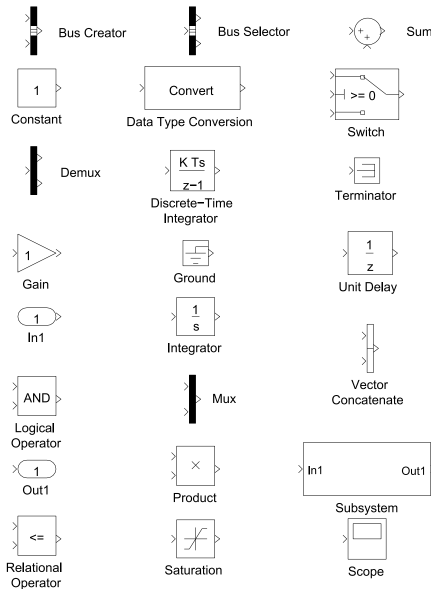

Abbildung 4.10: Ausschnitt der Simulink-Blockbibliothek (entnommen aus dem Simulink Library Browser).

In Abbildung 4.11 ist ein Minimalbeispiel eines Simulinkmodells bestehend aus acht Blöcken der Typen

- Constant,
- Outport,
- Relational Operator,
- Sum,
- Switch und
- UnitDelay

und neun Signalkanten gegeben. Die acht Blöcke werden nun zu den Knoten, die neun Signalkanten zu den gerichteten Kanten im Graphen. Der daraus resultierende Graph ist in Abbildung 4.12 dargestellt.

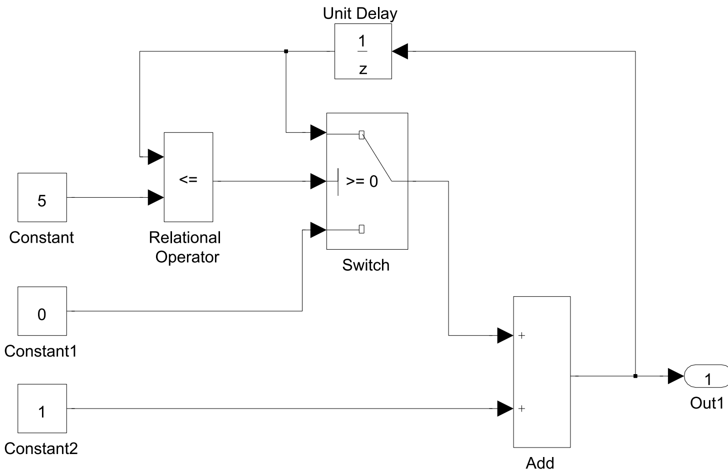

Abbildung 4.11: Beispielmodell.

4.2.3 Implementierung der Lernphase

In diesem Abschnitt wird die Implementierung der Pfadsuche, der Festlegung der Ursachenpfade und der nichtlinearen Optimierung beschrieben.

Pfadsuche

Die hier verwendete Pfadsuche beruht auf einer Tiefensuche (engl. depth-first search, DFS) in einem Graphen $G = (E, K)$, wie sie in Algorithmus 4.2 gegeben ist. Das Ziel dieser rekursiven Suche ist es, alle Knoten in dem Graphen zu finden. Einem Knoten $u \in E$ werden dafür vier Parameter zugewiesen:

- $u.\text{farbe} = \begin{cases} \text{weiss,} & \text{wenn der Knoten noch nicht besucht wurde} \\ \text{grau,} & \text{wenn der Knoten gerade bearbeitet wird} \\ \text{schwarz,} & \text{wenn der Knoten abgearbeitet ist} \end{cases},$

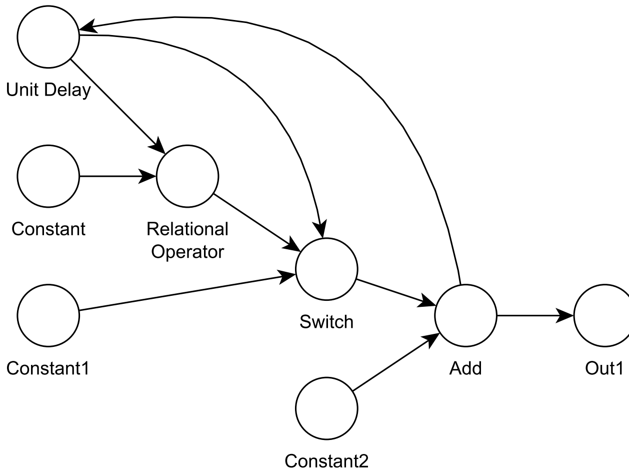

Abbildung 4.12: Graph entstehend aus dem Modell in Abbildung 4.11.

- $u.\pi$ = Vorgänger von u,
- u.d = Zeitpunkt des Entdeckens des Knotens,
- u.f = Zeitpunkt der Abarbeitung des Knotens.

In Abbildung 4.13 ist ein Graph dargestellt, auf den Algorithmus 4.2 angewendet wird. Die einzelnen Schritte sind in Tabelle 4.1 gezeigt. Die Reihenfolge der besuchten Knoten ist hier a, b, c gefolgt von d, e, f.

Für die Pfadsuche müssen die Algorithmen 4.2 und 4.3 modifiziert werden. Unterschiede zwischen der klassischen Tiefensuche und der Pfadsuche sind:

- Die Tiefensuche besucht alle Knoten im Graphen, während die Pfadsuche von einem Startknoten, dem Fehlerknoten, ausgeht und al-

Algorithmus 4.2 DFS(G)

1: **for** jeden Knoten $u \in E$ **do**
2: u.farbe = weiss // noch kein Knoten wurde besucht
3: $u.\pi$ = NIL // noch kein Knoten hat einen Vorgängerknoten
4: **end for**
5: zeit = 0 // initiale Zeit
6: **for** jeden Knoten $u \in E$ **do**
7: **if** u.farbe==weiss **then**
8: // Knoten wurde noch nicht besucht
9: DFS-Visit-Original(G,u)
10: **end if**
11: **end for**

le anderen Knoten, die nicht vom Startknoten aus erreicht werden können, ignoriert.

- In der Tiefensuche werden Kanten vorwärts verfolgt, d.h. ausgehend vom Knoten u werden alle seine Nachfolgerknoten $v_1, \ldots, v_n$, für die $(u, v_i) \in K$ für $i = 1, \ldots, n$ gilt, besucht. Bei der Pfadsuche erfolgt dies rückwärts, sodass die Vorgängerknoten besucht werden.

- Bei der Tiefensuche werden zu einem Knoten der Vorgängerknoten, die Färbung der Knoten, die Entdeckungszeit und die Abarbeitungs-

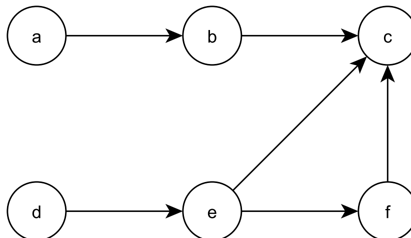

Abbildung 4.13: Graph für die Tiefensuche.

Algorithmus 4.3 DFS-Visit-Original(G,u)

```
1: zeit = zeit + 1
2: u.d = zeit // Entdeckungszeit
3: u.farbe = grau // Knoten ist aktuell in Bearbeitung
4: for jeden Knoten v ∈ Adj(u) do
5:    // alle adjazenten Knoten
6:    if v.farbe==weiss then
7:       // Knoten wurde noch nicht besucht
8:       v.π = u // u ist Vorgänger von v
9:       DFS-Visit-Original(G,v) // besuche v
10:   end if
11: end for
12: u.farbe = schwarz // u hat keine weiteren adjazenten Knoten
13: zeit = zeit + 1
14: u.f = zeit // Fertigstellungszeit von u
```

zeit gespeichert. Bei der Pfadsuche wird bei einem besuchten Knoten lediglich geprüft, ob er bereits im aktuellen Pfad enthalten ist. Falls dieser Fall eintritt, so wird der aktuelle Pfad gespeichert und der nächste Pfad bearbeitet. Wurde ein Knoten jedoch durch einen anderen Pfad besucht, so hat dieser Fall keine Auswirkungen auf die Tiefensuche. Deswegen wird die Färbung der Knoten nicht benötigt. Ebenfalls wird auf die Entdeckungs- und Abarbeitungszeit verzichtet. Des Weiteren wird nicht pro Knoten der Nachgängerknoten gespeichert, sondern der besuchte Knoten wird dem aktuellen Pfad hinzugefügt. Dazu wird die rückwärts gerichtete Tiefensuche soweit durchgeführt, bis ein Knoten keinen Vorgänger besitzt oder ein Abbruchkriterium erreicht wird. Zu diesem Zeitpunkt ist ein neuer Pfad vollständig gefunden, welcher der Gesamtmenge aller Pfade hinzugefügt wird. Mit jedem Backtracking in der Tiefensuche wird folglich ein neuer Pfad hinzugefügt.

Tabelle 4.1: Tiefensuche angewendet auf Abbildung 4.13. „far“ = Farbe, „w“= weiss, „g“= grau, „s“= schwarz, „N“ = NIL = noch nicht bestimmt

zeit	0	1	2	3	4	5	6	7	8	9	10	11	12
a.far	w	g	g	g	g	g	s	s	s	s	s	s	s
$a.\pi$	N	N	N	N	N	N	N	N	N	N	N	N	N
a.d	-	1	1	1	1	1	1	1	1	1	1	1	1
a.f	-	-	-	-	-	-	6	6	6	6	6	6	6
b.far	w	w	g	g	g	s	s	s	s	s	s	s	s
$b.\pi$	N	N	a	a	a	a	a	a	a	a	a	a	a
b.d	-	-	2	2	2	2	2	2	2	2	2	2	2
b.f	-	-	-	-	-	5	5	5	5	5	5	5	5
c.far	w	w	w	g	s	s	s	s	s	s	s	s	s
$c.\pi$	N	N	N	b	b	b	b	b	b	b	b	b	b
c.d	-	-	-	3	3	3	3	3	3	3	3	3	3
c.f	-	-	-	-	4	4	4	4	4	4	4	4	4
d.far	w	w	w	w	w	w	w	g	g	g	g	g	s
$d.\pi$	N	N	N	N	N	N	N	N	N	N	N	N	N
d.d	-	-	-	-	-	-	-	7	7	7	7	7	7
d.f	-	-	-	-	-	-	-	-	-	-	-	-	12
e.far	w	w	w	w	w	w	w	w	g	g	g	s	s
$e.\pi$	N	N	N	N	N	N	N	N	d	d	d	d	d
e.d	-	-	-	-	-	-	-	-	8	8	8	8	8
e.f	-	-	-	-	-	-	-	-	-	-	-	11	11
f.far	w	w	w	w	w	w	w	w	w	g	s	s	s
$f.\pi$	N	N	N	N	N	N	N	N	N	e	e	e	e
f.d	-	-	-	-	-	-	-	-	-	9	9	9	9
f.f	-	-	-	-	-	-	-	-	-	-	10	10	10

Der somit entstandene Pseudo-Code ist dargestellt in Algorithmus 4.4, welcher wiederum Algorithmus 4.5 aufruft.

Beispiel 4.1 (Pfadsuche)**.**
Angewendet auf das Modell in Abbildung 4.11 ergibt die Pfadsuche für

Algorithmus 4.4 Pfadsuche(G,s)

1: gefundenePfade = $\emptyset$ // noch keine Pfade gefunden
2: pfad = [s] // $s \in E$ Fehlerknoten
3: allePfade = DFS-Visit(G,s,pfad,gefundenePfade) // suche alle Pfade, die in s enden

den Startknoten `Out1` folgende Pfade beschrieben durch ihre Blocktypen:

p_1 : `Outport` → `Sum` → `Constant`,

p_2 : `Outport` → `Sum` → `Switch` → `Constant`,

p_3 : `Outport` → `Sum` → `Switch` → `Relational Operator` → `Constant`,

p_4 : `Outport` → `Sum` → `Switch` → `Relational Operator` → `Unit Delay` → `Sum` und

p_5 : `Outport` → `Sum` → `Switch` → `Unit Delay` → `Sum`.

Bei den Pfaden p_4 und p_5 wird das Abbruchkriterium „*Schleife gefunden*“ aktiv: in beiden Pfaden tritt der Block `Add` vom Typ `Sum` doppelt auf und die Pfade sind somit abgeschlossen. Die gefundenen Pfade werden in einer Datei in Form von Zeichenketten gespeichert, welche die Blocktypen in den jeweiligen Pfaden beschreiben.

Festlegung der Ursachenpfade

Um die Ursachen der Fehler der Lernphase in Simulink-/TargetLink-Modellen zu identifizieren, werden hier zwei Werkzeuge verwendet:

- Model Compare der Firma dSPACE (2015), und
- BTC EmbeddedValidator von BTC Embedded Systems AG (2014).

Im Model Compare von dSPACE werden zwei aufeinanderfolgende Versionen eines Modells verglichen, um Unterschiede aufzudecken. Beispiele für Unterschiede sind:

Algorithmus 4.5 DFS-Visit(G,u,pfad,gefundenePfade)

```
1:  // G: Graph
2:  // u: aktuell besuchter Knoten
3:  // pfad: aktueller Pfad
4:  // gefundenePfade: Auflistung aller bisher gefundenen Pfade
5:  if Adj(u) == ∅ then
6:     // u hat keine adjazenten Knoten
7:     gefundenePfade = {gefundenePfade,pfad}
8:     // füge aktuellen Pfad zu der Menge aller bisher gefundenen Pfade
       hinzu
9:  else
10:    //u hat adjazente Knoten
11:    v = Adj(u)
12:    for i=1 to length(v) do
13:       if v(i)∉ pfad then
14:          // v(i) nicht im aktuellen Pfad enthalten
15:          gefundenePfade = DFS-Visit(G,v(i),[pfad,v(i)],gefundenePfade)

16:          // besuche aktuellen adjazenten Knoten
17:       else
18:          // v(i) im aktuellen Pfad enthalten
19:          gefundenePfade = {gefundenePfade,[pfad,v(i)]}
20:       end if
21:    end for
22: end if
```

- hinzugefügte oder gelöschte Blöcke,

- hinzugefügte oder gelöschte Signallinien,

- Veränderung von Blockparametern (Wert im `Constant`-Block, Faktor im `Gain`-Block, Auswahl der Einzelsignale in Bussignalen, usw.) oder

- Veränderung von Signalen (Start- und Endblöcke, Nummer der Ports an den verbundenen Blöcken, usw.).

Relevant für die Suche nach den Ursachenpfaden sind hier die Unterschiede zwischen den beiden Modellversionen, die im Signal- bzw. Datenfluss vor dem Fehlerblock gefunden werden. Zur Beurteilung, ob der jeweilige Unterschied für den Fehler verantwortlich gemacht werden kann, werden die Unterschiede über ein manuelles Review verglichen. Im Rahmen dieses Reviews muss ebenfalls festgelegt werden, welcher Umfang genau zu dem verantwortlichen Unterschied gehört. Der Model-Checker BTC Embedded-Validator der BTC Embedded Systems AG wird eingesetzt, wenn es keine Nachfolgeversion des aktuellen Modells gibt, in welcher der Fehler nicht enthalten ist, oder wenn der Vergleich der Modellversionen nicht weiterhelfen konnte. Abbildung 4.14 zeigt den Prozess zur Verwendung des Model-

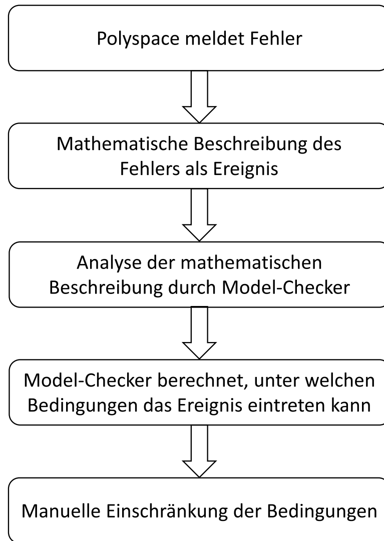

Abbildung 4.14: Verwendung des Model-Checkers für die Ursachenidentifikation.

Checkers. Der durch Polyspace analysierte Fehler muss zunächst mathematisch beschrieben werden. Zum Beispiel sei das Ergebnis der Polyspace-Analyse ein Overflow für die Code-Zeile `x = x + 1`, wobei `x` vom Typ `char` ist. Eine mögliche mathematische Beschreibung für das durch den Model-Checker zu prüfende Ereignis ist `x > 127`. Diese Beschreibung wird im nächsten Schritt durch den Model-Checker analysiert. Dabei berechnet der Model-Checker, unter welchen Bedingungen das Ereignis eintreten kann. Diese Bedingungen werden durch ein manuelles Review eingeschränkt, um die tatsächlichen Ursachenpfade zu erhalten.

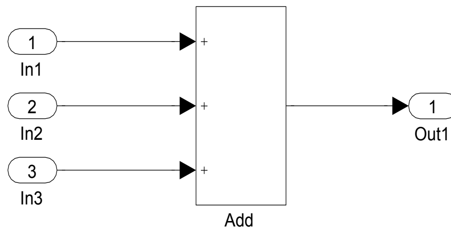

Abbildung 4.15: Subsystem für die Anwendung des Model-Checkers.

In Abbildung 4.15 ist ein Subsystem eines Minimalbeispiels in Simulink gegeben. Der Block `Add` vom Typ `Sum` ist vom Datentyp `Int8`, d.h. innerhalb des Intervalls $[-128, 127]$. Die Polyspace-Analyse ergibt für den `Sum`-Block einen Overflow über die obere Grenze hinaus. Bezüglich der Signale `In1`, `In2` und `In3` ist folgendes bekannt:

- jedes der Signale ist ein Vektor der Größe 6,
- der i-te Eintrag ist bei genau einem Vektor ungleich Null und
- jeder Vektor hat genau zwei Einträge ungleich Null.

Mittels des Model-Checkers werden nun drei Ereignisse analysiert:

$E_1 =$ `(In1[0] > 127) || (In1[1] > 127) || (In1[2] > 127) ||`
`(In1[3] > 127) || (In1[4] > 127) || (In1[5] > 127)`,

$E_2 =$ `(In2[0] > 127) || (In2[1] > 127) || (In2[2] > 127) ||`
`(In2[3] > 127) || (In2[4] > 127) || (In2[5] > 127)` und

$E_3 =$ `(In3[0] > 127) || (In3[1] > 127) || (In3[2] > 127) ||`
`(In3[3] > 127) || (In3[4] > 127) || (In3[5] > 127)`.

Die Analyse ergibt, dass die Ereignisse E_1 und E_3 nicht erreichbar sind. Ereignis E_2 kann erreicht werden. Aus diesem Grund können die Vorgänger von `In1` und `In3` vernachlässigt und nur `In2` zurück verfolgt werden.

```
Selector      Outport Switch Inport Logic  Logic
    RelationalOperator   BusSelector
Selector      Outport Switch Inport Logic  Logic
    RelationalOperator   UnitDelay
U: Selector   Outport Switch Inport Switch Constant
Selector      Outport Switch Inport Switch Logic  BusSelector
        Inport From
```

Code 4.3: Beispielmarkierung eines gefundenen Ursachenpfades.

Die durch die beschriebenen Methoden ermittelten Ursachenpfade werden in den Textdateien, welche die Pfade enthalten, durch den String „U:“ markiert, wie in Code 4.3 dargestellt.

Nichtlineare Optimierung

Zum Lösen des nichtlinearen Optimierungsproblems

$$\min_{\mathbf{w}\in\mathbb{R}^n} \{f(\mathbf{w}) \mid \mathbf{Sw} \leq \mathbf{o}\}$$

bzw. des Problems

$$\min_{\mathbf{w} \in D} f(\mathbf{w})$$

mit $f(\mathbf{w}) = (\mathbf{Rw})_{\max} - (\mathbf{Uw})_{\min}$ wird die *Optimization Toolbox* von Matlab verwendet. MathWorks (2015c) beschreibt, dass die Funktion `fmincon` das nichtlineare Optimierungsproblem

$$\min_{\mathbf{x}} \{f(\mathbf{x})\}$$

unter den Nebenbedingungen

$$\begin{aligned} \mathbf{c}(\mathbf{x}) &\leq \mathbf{0}, \\ \mathbf{c}_{eq}(\mathbf{x}) &= \mathbf{0}, \\ \mathbf{A} \cdot \mathbf{x} &\leq \mathbf{b}, \\ \mathbf{A}_{eq} \cdot \mathbf{x} &= \mathbf{b}_{eq} \text{ und} \\ \mathbf{lb} \leq \mathbf{x} &\leq \mathbf{ub} \end{aligned}$$

löst. Hierbei können die Funktionen $f, \mathbf{c}$ und $\mathbf{c}_{eq}$ nichtlinear sein. Angewendet auf die Optimierung für diese Arbeit bleiben folgende Restriktionen übrig:

$$\begin{aligned} \mathbf{A} \cdot \mathbf{x} &\leq \mathbf{b} \text{ und} \\ \mathbf{lb} \leq \mathbf{x} &\leq \mathbf{ub}. \end{aligned}$$

Die Verwendung der ersten Restriktion führt zu dem Optimierungsproblem in (3.12), die Verwendung der zweiten entspricht dem Problem in (3.11). Beide Arten der Restriktionen liefern denselben Zielfunktionswert zurück, deswegen wird nachfolgend aus Gründen der Einfachheit nur die Form aus (3.12) verwendet.

Die Funktion `fmincon` kann vier verschiedene Algorithmen verwenden, von

denen genau einer ausgewählt wird:

- `interior-point`: bewältigt große, dünn besetzte Probleme und erfüllt die Grenzen während jeder Iteration
- `trust-region-reflective`: benötigt die Angabe des Gradienten und erlaubt entweder angegebene Grenzen oder lineare Gleichungsrestriktionen. Der Algorithmus kann sowohl große, dünn besetzte als auch kleine, vollbesetzte Probleme lösen
- `sqp` (*sequential quadratic programming*): erfüllt die Grenzen während jeder Iteration
- `active-set`: kann mit großen Schrittweiten umgehen, um die Geschwindigkeit zu erhöhen und ist effektiv für Probleme mit nichtglatten Restriktionen.

Diese Informationen sind MathWorks (2015a) entnommen. Wird keiner der Algorithmen explizit angegeben, so entscheidet Matlab selbstständig, welcher Algorithmus für das gegebene Problem geeignet ist. Für das in dieser Arbeit beschriebene Optimierungsproblem wird der *active-set*-Algorithmus gewählt. Dazu wird die Lagrange-Funktion aus (3.24) verwendet. Anhand dieser Funktion wird laut MathWorks (2015b) ein quadratisches Subproblem gebildet, welches den Gradienten der Zielfunktion sowie die Hesse-Matrix der Lagrange-Funktion verwendet. Der Gradient und die Hesse-Matrix werden dabei gemäß Lee (2009) durch finite Differenzen approximiert. Die Tatsache, dass die Zielfunktion aus mathematischer Sicht nicht eindeutig differenzierbar ist, wird durch diese Approximation vernachlässigt. Aus diesem Grund wird durch Matlab eine andere Optimierungsmethode verwendet als in Kapitel 3.2 beschrieben. Aus mathematischer Sicht wären die dort erläuterten Algorithmen angemessen, um eine Garantie der Konvergenz zu bekommen. Jedoch sind die beschriebenen Algorithmen nicht in der verwendeten Toolbox von Matlab verfügbar.

Die hier präsentierte Methode ist heuristikbasiert, sodass die Optimierung nicht konvergieren muss. Durch die Approximation besteht hier das Risiko, dass die Optimierung fehlschlagen kann. Dieses ist für die Daten in der vorliegenden Arbeit nicht eingetreten. Da die Optimierung nicht zeitkritisch ist, wird auf die Bestimmung guter Startpunkte verzichtet.

Beschreibung der verwendeten Daten

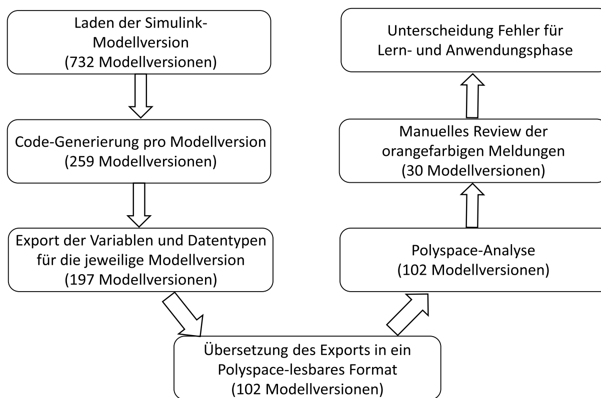

Abbildung 4.16: Prozess zur Identifikation der Fehler.

Zur Ausführung der Lernphase müssen die zu betrachtenden Fehler identifiziert werden. Dazu wird der Prozess verfolgt, wie er in Abbildung 4.16 beschrieben ist. Im ersten Schritt wird jede der gegebenen 732 Modellversionen in der Matlab-Umgebung geladen. Für jede Modellversion wird im Anschluss versucht, C-Code zu generieren. Diese Generierung ist für 259 Modellversionen erfolgreich. Der hohe Datenverlust ist dadurch begründet, dass es sich bei den Modellversionen um Zwischenstände im Entwicklungsstadium handelt. In den Modellen sind Variablen und Datentypen explizit beschrieben. Diese Daten werden exportiert (erfolgreich für 197 Modellversionen) und anschließend in ein durch Polyspace lesbares Format übersetzt.

Die Übersetzung erfolgt für 102 Modellversionen. Der generierte Code dieser 102 Modellversionen wird im nächsten Schritt zusammen mit den exportierten Daten durch Polyspace analysiert. In 30 dieser Analysen gibt Polyspace Meldungen aus, die nachfolgend einem manuellen Review unterzogen werden. Betrachtet werden Fehler vom Typ *Overflow* und *Array out of Bounds* (Zugriff auf ein Array außerhalb der definierten Größe), da diese Fehlertypen in der PKW-Entwicklung bei der Daimler AG am häufigsten auftreten. Der letzte Schritt in diesem Prozess ist die Unterscheidung, welche der insgesamt 54 identifizierten Fehler für die Lern- und welche für die Anwendungsphase verwendet werden. Dabei werden 35 Fehler beliebig für die Lernphase gewählt. Zu diesen 35 Fehlern führen über 3500 Pfade, wovon 107 überapproximiert als Ursachenpfade gewertet werden. Die Pfade enthalten 22 verschiedene Blocktypen:

- `ActionPort`,
- `Assignment`,
- `BusCreator`,
- `BusSelector`,
- `Constant`,
- `Demux`,
- `ForIterator`,
- `From`,
- `Gain`,
- `Ground`,
- `RelationalOperator`,
- `Inport`,
- `Logic`,
- `Merge`,
- `MultiPortSwitch`,
- `Outport`,
- `Product`,
- `If`,
- `Selector`,
- `Sum`,
- `Switch`,
- `UnitDelay`.

Der gesuchte Gewichtevektor hat somit die Dimension 22 und die Matrizen $\mathbf{U}$ und $\mathbf{R}$ bestehen jeweils aus 22 Spalten. Nach Löschung von redundanten Pfaden ergibt sich für die Größe der Matrizen $\mathbf{U} \in \mathbb{R}^{102\times 22}$ und $\mathbf{R} \in \mathbb{R}^{3494\times 22}$. Für die Berechnung des Gewichtevektors $\mathbf{w} \in \mathbb{R}^{22}$ gelten bei der Optimierung Toleranzgrenzen von 10^{-10} und maximal 50.000 Iterationen

sollen durchgeführt werden. Als Infimum des Gewichtevektors $\mathbf{w}$ wird 0 angenommen, für das Supremum wird 100 gewählt:

$$0 \leq w_i \leq 100 \;\forall\; 1 \leq i \leq 22.$$

Daraus ergibt sich die Matrix

$$\mathbf{S} = \begin{pmatrix} -1 & & & \\ & -1 & & \\ & & \ddots & \\ & & & -1 \\ 1 & & & \\ & 1 & & \\ & & \ddots & \\ & & & 1 \end{pmatrix} \in \mathbb{R}^{44 \times 22}$$

und der Vektor

$$\mathbf{o} = \begin{pmatrix} 0 & 0 & \dots & 0 & 100 & 100 & \dots & 100 \end{pmatrix}^\top \in \mathbb{R}^{44}.$$

Die Ungleichung $\mathbf{Sw} \leq \mathbf{o}$ beschreibt somit die angegebenen Grenzen. Das zu lösende Optimierungsproblem hat die Form

$$\min_{\mathbf{w} \in \mathbb{R}^{22}} \{ f(\mathbf{w}) \mid \mathbf{Sw} \leq \mathbf{o} \}$$

mit

$$f(\mathbf{w}) = (\mathbf{Rw})_{\max} - (\mathbf{Uw})_{\min}.$$

Matlab liefert für das Problem als Ergebnis den Gewichtevektor

$$\mathbf{w} = \begin{pmatrix} 0,0987 \\ 100,0000 \\ 92,7586 \\ 0 \\ 40,5744 \\ 0,7212 \\ 37,6717 \\ 31,0286 \\ 29,2175 \\ 0,7791 \\ 0,7150 \\ 76,5775 \\ 43,8641 \\ 0,3342 \\ 3,1606 \\ 50,5266 \\ 37,9541 \\ 22,7091 \\ 86,5266 \\ 73,9889 \\ 61,3162 \\ 31,7659 \end{pmatrix} \tag{4.2}$$

zurück. Dabei werden für die Erstellung der Matrizen **R** und **U** sowie die Berechnung der Lösung des Optimierungsproblems ungefähr 110 Minuten benötigt. [1]

[1] auf einem handelsüblichen Notebook der Firma Dell mit Intel(R) Core (TM) i5-4310M CPU @ 2,70GHz und einem Hauptspeicher von 8094 MB

4.2.4 Implementierung der Anwendungsphase

Für die Anwendungsphase wird ein Testset mit 19 Fehlern vom Typ *Overflow* und *Array out of Bounds* verwendet. Zu diesen 19 Fehlern führen 2974 Pfade, wovon zwei redundant sind. Nachdem die Fehler und die entsprechenden Pfade zu den Fehlern bestimmt sind, werden die Pfade durch ihre Blocktypen beschrieben und in einer Textdatei gespeichert. Zu den Fehlern werden manuell insgesamt 46 potentielle Ursachenpfade überapproximiert und in der Textdatei mittels „U:“ markiert. Ziel bei dieser Markierung ist es, dass nach Abschließen der Anwendungsphase geprüft werden kann, wie viele Ursachenpfade tatsächlich durch das Verfahren getroffen wurden. Im nächsten Schritt wird pro Fehler die Matrix $\mathbf{A}$ erstellt, welche die Blocktypen in den Pfaden beschreibt. Aus den 19 Fehlern resultieren demnach 19 Matrizen:

1. $\mathbf{A}_1 \in \mathbb{R}^{628\times22}$,
2. $\mathbf{A}_2 \in \mathbb{R}^{628\times22}$,
3. $\mathbf{A}_3 \in \mathbb{R}^{76\times22}$,
4. $\mathbf{A}_4 \in \mathbb{R}^{75\times22}$,
5. $\mathbf{A}_5 \in \mathbb{R}^{75\times22}$,
6. $\mathbf{A}_6 \in \mathbb{R}^{75\times22}$,
7. $\mathbf{A}_7 \in \mathbb{R}^{75\times22}$,
8. $\mathbf{A}_8 \in \mathbb{R}^{119\times22}$,
9. $\mathbf{A}_9 \in \mathbb{R}^{118\times22}$,
10. $\mathbf{A}_{10} \in \mathbb{R}^{118\times22}$,
11. $\mathbf{A}_{11} \in \mathbb{R}^{119\times22}$,
12. $\mathbf{A}_{12} \in \mathbb{R}^{118\times22}$,
13. $\mathbf{A}_{13} \in \mathbb{R}^{8\times22}$,
14. $\mathbf{A}_{14} \in \mathbb{R}^{8\times22}$,
15. $\mathbf{A}_{15} \in \mathbb{R}^{8\times22}$,
16. $\mathbf{A}_{16} \in \mathbb{R}^{8\times22}$,
17. $\mathbf{A}_{17} \in \mathbb{R}^{8\times22}$,
18. $\mathbf{A}_{18} \in \mathbb{R}^{355\times22}$ und
19. $\mathbf{A}_{19} \in \mathbb{R}^{355\times22}$.

Um die Pfadgewichte zu berechnen, wird nun jede Matrix $\mathbf{A}_i$, $1 \leq i \leq 19$,

mit dem Gewichtevektor aus (4.2) multipliziert. Von dem Ergebnis dieser Multiplikation werden nun diejenigen Indizes gewählt, deren Eintrag unter den $c \in \mathbb{N}$ höchsten Werten liegt. Es kann vorkommen, dass es mehrere Einträge mit dem Wert x gibt, sodass die Anzahl der gewählten Indizes größer als c sein kann. Die Pfade zu diesen Indizes werden in der Textdatei durch „*“ gekennzeichnet. Zur Überprüfung des Erfolgs der beschriebenen Methode wird geprüft, wie viele Pfade es gibt, die durch die Kombinationsmarkierung „*U:“ gekennzeichnet sind. Ein Beispiel dazu ist in Code 4.4 gegeben.

```
Selector      Outport Switch Inport Logic  Logic
    RelationalOperator   BusSelector
Selector      Outport Switch Inport Logic  Logic
    RelationalOperator   UnitDelay
*U: Selector  Outport Switch Inport Switch Constant
Selector      Outport Switch Inport Switch Logic  BusSelector
       Inport From
```

Code 4.4: Beispielmarkierung eines berechneten Ursachenpfades.

4.3 Evaluierung der Ergebnisse

Nach der Multiplikation der 19 Matrizen $\mathbf{A}_1$ bis $\mathbf{A}_{19}$ mit dem Gewichtevektor aus (4.2) werden diese nach absteigendem Pfadgewicht sortiert. Die natürliche Zahl c bezeichne die Anzahl der betrachteten Pfadgewichte. Als Beispiel ergibt die Multiplikation von $\mathbf{A}_{13}$ mit $\mathbf{w}$ die Pfadgewichte, wie sie in Tabelle 4.2 dargestellt sind. Die Sortierung ergibt die in Tabelle 4.3 angegebene Reihenfolge. Für $c = 1$ wird der Pfad p_6 mit dem höchstem Gewicht gewählt, für $c = 2$ sind es die Pfade mit dem höchstem und zweithöchstem Gewicht, also p_6 und p_7, und für $c = 4$ werden alle Pfade $p_1, \ldots, p_8$ gewählt.

Tabelle 4.2: Beispielpfadgewichte.

Pfad	Pfadgewicht
p_1	62,1490
p_2	62,1490
p_3	62,1490
p_4	62,1490
p_5	62,1490
p_6	72,0966
p_7	66,5386
p_8	62,8057

Die Evaluierung beginnt jeweils mit $c = 1$. Dadurch werden für alle 19 Fehler insgesamt 27 Pfade markiert, von denen 17 zuvor als Ursachenpfade markiert wurden. Zur Erinnerung: die Anzahl aller markierten Ursachenpfade beträgt 46. Für $c = 1$ wird somit eine Trefferquote von $17/46 \approx 37\%$ erreicht, wobei zwei Fehler dadurch bereits komplett abgedeckt sind. Um die verbleibenden 29 Ursachenpfade zu erhalten, wird c nun auf 2 erhöht. Dadurch werden weitere Pfade markiert, sodass sich die Anzahl der Reviewpfade nun auf einen Wert von 52 beläuft. Die Trefferquote liegt hier

Tabelle 4.3: Sortierte Beispielpfadgewichte.

Pfad	Pfadgewicht
p_6	72,0966
p_7	66,5386
p_8	62,8057
p_1	62,1490
p_2	62,1490
p_3	62,1490
p_4	62,1490
p_5	62,1490

bei 69,6% und Ursachenpfade für sieben Fehler sind gefunden. Für $c = 3$ liegt die Trefferquote bereits bei 91,3%. Hier sind die Ursachenpfade für alle Fehler mit Ausnahme von zweien gefunden worden. Zu diesen zwei Fehlern führen jeweils über 600 Pfade. Hier wird mit $c = 9$ eine Trefferquote von 95,7% für alle 19 Fehler und mit $c = 10$ eine Trefferquote von 100% erreicht. Für die 100% Trefferquote werden insgesamt 158 von den knapp 3000 Pfaden markiert, also ca. 5,3% der Pfade. Diese Daten sind in Tabelle 4.4 noch einmal verkürzt dargestellt.

Für die Erstellung der Matrix in der Anwendungsphase, die Gewichtung und die Sortierung der Pfade für alle 19 Fehler werden für $c = 1, \ldots, 10$ im Durchschnitt nur 0,63 Sekunden benötigt. [1] Der Zeitaufwand ist abhängig von dem Wert c und ist in Tabelle 4.4 in der zweiten Spalte beschrieben. Die dritte Spalte beschreibt die Anzahl der gefundenen Reviewpfade, der Anteil davon bezogen auf die Gesamtanzahl der Pfade ist angegeben in der vierten Spalte. In der fünften Spalte wird berechnet, wie viele Ursachenpfade bezogen auf die Gesamtanzahl aller Ursachenpfade durch die Methode identifiziert wurden. Die letzte Spalte vergleicht den manuellen Reviewaufwand mit einem von Experten der Daimler AG ermittelten Erfahrungswert. Im Gegensatz zu einem Entwickler führen diese Experten täglich Analysen durch und beschäftigen sich mit der Identifizierung der Ursachen der vorhandenen Fehler. Demnach müssen im Durchschnitt ca. 10 Pfade pro Fehler einem Review unterzogen werden. Die hier betrachtete Fehlerzahl liegt bei 19, deswegen beschreibt der Erfahrungswert 190 Pfade. Bezogen auf die Gesamtzahl aller Pfade wird hier der Anteil $190/2974 = 6{,}4\%$ betrachtet. Diese Ergebnisse sind ebenfalls graphisch dargestellt in Abbildung 4.17. Für eine vollständige Identifikation müssen durch die hier präsentierte Methode nur 5,3% aller möglichen Pfade einem

[1] auf einem handelsüblichen Notebook der Firma Dell mit Intel(R) Core (TM) i5-4310M CPU @ 2,70GHz und einem Hauptspeicher von 8094 MB

Tabelle 4.4: Ergebnisse der Evaluierung für die Fehlerarten `Overflow` und `Array out of Bounds`.

c	Zeit	Anzahl Reviewpfade	Reviewpfade/ Gesamtpfade	gefundene Ursachenpfade/ Gesamtursachen	Reviewaufwand/ Erfahrungswert
1	0,648s	27	0,9%	37%	0,9/6,4 = 14,1%
2	0,616s	52	1,7%	69,6%	1,7/6,4 = 26,6%
3	0,604s	72	2,4%	91,3%	2,4/6,4 = 37,5%
9	0,642s	126	4,2%	95,7%	4,2/6,4 = 65,5%
10	0,633s	158	5,3%	100%	5,3/6,4 = 82,8%

manuellen Review unterzogen werden. Dies sind 82,8% von dem durchschnittlichen Erfahrungswert der Experten der Daimler AG, sodass der Aufwand der Experten um ca. 17% reduziert werden kann. Der manuelle Reviewaufwand wird hier zusätzlich optimiert, da die Reviewpfade im Modell zurückverfolgt werden können. Dabei kann den Signallinien im Modell gefolgt werden. Dagegen müssen für die Verfolgung im Code die jeweiligen Variablen- bzw. Funktionsnamen gesucht werden. Hier ist zu beachten, dass der Erfahrungswert von Experten auf dem Gebiet von Software-Analysen stammt. Der Reviewaufwand eines Entwicklers ist höher, da er mehr Aufwand in die Einschätzung der Fehler und der Ursachen investieren muss. Verglichen dazu wird der Aufwand durch die in dieser Arbeit präsentierte Methode um weit mehr als 17% reduziert.

Dasselbe Verfahren wurde im Vorfeld durch Schneider (2014) bereits auf

Tabelle 4.5: Ergebnisse für die Polyspace-Meldung *Unreachable Code*.

c	Anzahl Reviewpfade	Reviewpfade/ Gesamtpfade	gefundene Ursachenpfade/ Gesamtursachen	Reviewaufwand/ Erfahrungswert
1	54	0,11%	83,3%	0,11/1,12 = 9,8%
2	63	0,13%	90,7%	0,13/1,12 = 11,6%
3	68	0,14%	100%	0,14/1,12 = 12,5%

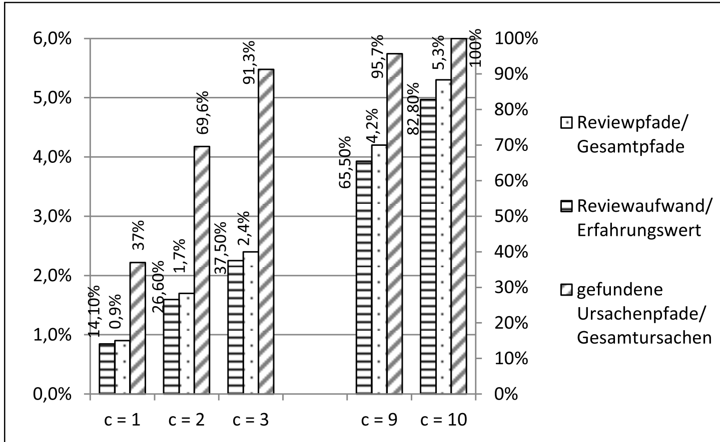

Abbildung 4.17: Evaluierung der angewendeten Methode für die Fehlerarten `Overflow` und `Array out of Bounds`.

den Polyspace-Ergebnistyp *Unreachable Code* angewendet, d.h. Code-Abschnitte, die nicht erreichbar sind. Ein Beispiel für diese Situation ist in Abbildung 4.18 dargestellt. In Abbildung 4.18 wird links das `Subsystem` aufgerufen mit einem konstanten Wert für den Eingang `E`. Die Kennzeichnung `True` symbolisiert hier den Wert `1`. Das aufgerufene `Subsystem` ist rechts in Abbildung 4.18 dargestellt. Der Eingang `E` dient als Entscheidungssignal für den Switch-Block `Switch_C`. Die Entscheidung in dem Switch-Block ist `E` $\geq$ `1` und somit mit `E=1` immer erfüllt. Für die Bearbeitung der Kategorie *Unreachable Code* werden 33 Blocktypen anhand von 27 Polyspace-Meldungen gewichtet, zu denen 19.328 Pfade führen. Die Anwendungsphase wird anhand von 129 Meldungen mit insgesamt über 48.000 Pfaden evaluiert. Von den 129 Meldungen gibt es 75, zu denen lediglich ein Pfad führt. Da der Ursachenpfad hier bereits vor Anwendung des Verfahrens eindeutig ist, werden diese Meldungen vernachlässigt. So-

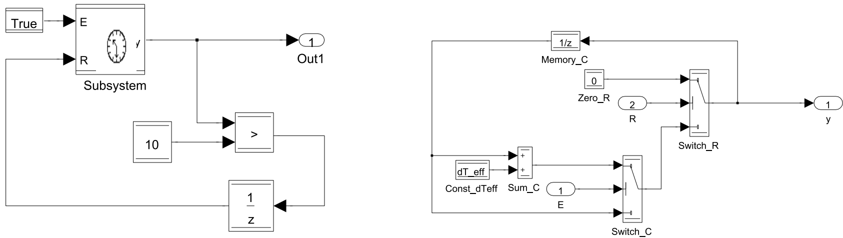

Abbildung 4.18: Links: Modell, rechts: `Unreachable Code` im `Subsystem`.

mit verbleiben 54 Meldungen mit insgesamt 48251 Pfaden.

Tabelle 4.5 zeigt die Ergebnisse des Verfahrens. Die Markierung derjenigen Pfade mit höchstem Gewicht, also $c = 1$, führt zu 54 Reviewpfaden und ergibt eine Trefferquote von 83,3%. Die Verwendung von $c = 2$ deckt mit 63

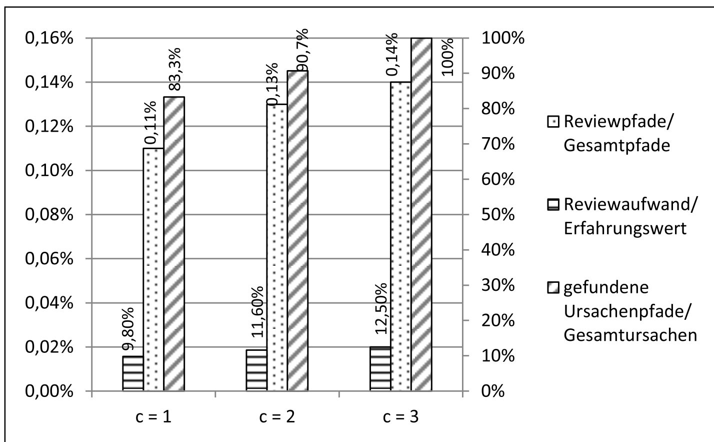

Abbildung 4.19: Evaluierung der angewendeten Methode für *Unreachable Code*.

Reviewpfaden bereits 90,7% aller Ursachenpfade ab. Mit nur $c = 3$ werden alle Ursachenpfade gefunden. Hierbei wurden 68 Reviewpfade markiert. Insgesamt betrug die Menge der Reviewpfade 0,14% von allen Pfaden, die zu allen Meldungen führen. Diese Ergebnisse sind in Abbildung 4.19 graphisch dargestellt. Der Erfahrungswert von 10 Pfaden pro Meldung ergibt hier $54*10 = 540$ Reviewpfade, also 1,12% der Gesamtpfade. Um alle Ursachen durch die präsentierte Methode zu identifizieren, werden nur 12,5% des manuellen Reviewaufwands verglichen zum Erfahrungswert benötigt.

Die Ergebnisse der Evaluierung zeigen, dass das Verfahren für die genannten Fehlerarten *Overflow* und *Array-out-of-Bounds* sowie für die Meldung *Unreachable Code* erfolgreich ist. Der dabei entstehende manuelle Reviewaufwand liegt signifikant unterhalb der Schätzung, wie sie von Experten der Daimler AG angegeben wird. Dadurch konnte gezeigt werden, dass das in dieser Arbeit präsentierte Verfahren nicht nur für das verwendete Evaluierungs-Set anwendbar ist, sondern dass es den manuellen Aufwand tatsächlich reduziert.

5 Zusammenfassung und weitere Arbeiten

5.1 Zusammenfassung der Ergebnisse

In der vorliegenden Arbeit wurde ein Verfahren zur Minimierung des manuellen Aufwands bei der Ursachenidentifikation von Laufzeitfehlern in Software präsentiert. Für die Korrektur der Laufzeitfehler müssen die Ursachen identifiziert werden. Um den manuellen Aufwand dabei zu reduzieren, wird ein teilautomatisiertes Verfahren präsentiert. Dieses Verfahren besteht aus zwei Phasen, einer Lern- und einer Anwendungsphase. In der Lernphase wird eine Menge an Fehlern verwendet, welche in der vorliegenden Dissertation durch Fehler in Simulink-/TargetLink-Modellen repräsentiert wird. Für diese Fehler werden alle Pfade ermittelt, die zu dem jeweiligen Fehler führen. Dabei werden die Ursachenpfade markiert, sodass zwei Matrizen berechnet werden können. Eine Matrix beschreibt die Ursachenpfade ($\mathbf{U}$) und die andere die verbleibenden Pfade ($\mathbf{R}$). Dabei repräsentiert eine Matrixzeile einen Pfad, eine Spalte einen spezifizierten Blocktyp. Alle gegebenen Blocktypen werden gewichtet, indem im Rahmen einer nichtlinearen Optimierung der Gewichtevektor $\mathbf{w}$ berechnet wird. Als

Zielfunktion der Optimierung wird die Funktion $(\mathbf{Rw})_{\max} - (\mathbf{Uw})_{\min}$ verwendet. Dieser Gewichtevektor ist das Ergebnis der Lernphase und dient als Eingabe für die Anwendungsphase. Die Anwendungsphase wird einmal pro Fehler durchgeführt. Hier werden ebenfalls alle Pfade ermittelt, die zum jeweiligen Fehler führen. Jeder der Pfade wird anhand der Blockgewichte aus der Lernphase gewichtet. Je größer das Pfadgewicht, desto höher ist die Wahrscheinlichkeit, dass der Pfad die Ursache des Fehlers enthält.

Um die entwickelte Heuristik zu evaluieren, wurde für die Lernphase ein Fehlerset von 35 Fehlern vom Typ *Overflow* und *Array out of Bounds* verwendet. Für die Anwendungsphase wurde ein von der Lernphase disjunktes Set von 19 Fehlern eingesetzt. Dabei konnte gezeigt werden, dass die in dieser Arbeit beschriebene Methode den manuellen Reviewaufwand reduziert. Als Vergleichswert dient der Erfahrungswert von Experten der Daimler AG, die im Gegensatz zu einem Entwickler tagtäglich Analysen durchführen und Ursachen für die Fehler identifizieren. Der berechnete Reviewaufwand beträgt 82,8% von dem Erfahrungswert der Experten für die Fehlerarten *Overflow* und *Array out of Bounds*. Für die Identifizierung der Ursachen der Meldung *Unreachable Code* werden sogar nur 12,5% von dem Erfahrungswert benötigt. Die Ursachenidentifikation bedeutet für einen Entwickler enormen Mehraufwand als für die Experten. Deswegen ist anzunehmen, dass die präsentierte Methode den Aufwand eines Entwicklers signifikant reduziert.

5.2 Vorschläge für weitere Forschungsarbeiten

Im Abschnitt *Nichtlineare Optimierung* wird erklärt, dass Matlab im Rahmen der nichtlinearen Optimierung den Gradienten der Zielfunktion sowie die Hesse-Matrix der Lagrangefunktion approximativ berechnet. Zur Ver-

besserung könnte hier ein Optimierungsverfahren entwickelt werden, welches die analytisch berechneten Gradienten und Hesse-Matrizen benutzt. Die daraus resultierenden genaueren Werte des Gewichtevektors führen wahrscheinlich zu einer verbesserten Unterscheidung der Ursachenpfade und der verbleibenden Pfade während der Anwendungsphase.

Die beschriebenen Methoden wurden für diese Arbeit auf die Fehlertypen *Overflow* und *Array out of Bounds* angewendet, da dies die am häufigsten vorkommenden Fehlertypen in der bereitgestellten Software der Daimler AG sind. Zusätzlich dazu wurde die Polyspace-Meldung *Unreachable Code* behandelt. Polyspace ist jedoch in der Lage, noch weitere Fehlertypen während der Analysen zu finden. Durch eine Vergrößerung der Fehlermenge können diese Typen ebenfalls behandelt werden.

Neben dieser Erweiterung ist auch eine Optimierung denkbar: eine Prüfung auf eine mögliche Abhängigkeit von Fehlern vor der Anwendung der Methode. Seien f_1 und f_2 zwei Fehler, wobei f_1 im Datenfluss vor f_2 auftritt. An der Stelle kann geprüft werden, ob f_2 von f_1 abhängig ist. Ein möglicher Hinweis für diese Abhängigkeit ist es, wenn ein Pfad existiert, der sowohl f_1 als auch f_2 enthält. Hier wäre der Optimierungsvorschlag, dass zunächst f_1 korrigiert wird und im Anschluss analysiert wird, ob die Korrektur den Fehler f_2 ebenfalls behoben hat. Dadurch würde die Ursachensuche für f_2 vermieden werden.

Denkbar wäre es ebenfalls, dass die präsentierte Methode bei dem Review der orangefarbenen Polyspace-Meldungen unterstützen kann. Eine mögliche Hypothese hier wäre: enthalten die ersten x Reviewpfade keine erkennbare Ursache, so handelt es sich sehr wahrscheinlich um einen *false positive*.

Die in dieser Arbeit beschriebene Evaluierung bezieht sich auf Simulink-/TargetLink-Modelle. Aktuell ist keine Einschränkung erkennbar, wieso diese Methodik nicht ebenfalls auf Code in diversen Sprachen anwendbar sein könnte. Als Beispiele für Knotengewichte könnten hier Operatoren wie Addition, Subtraktion usw. dienen.

Ein weiterer Schritt in der Forschung könnte darin bestehen, die durch die präsentierte Methode eingegrenzten Ursachen automatisiert zu korrigieren. Dazu könnte ermittelt werden, welche Ursachen für welche Fehlertypen gehäuft auftreten, um vielleicht ein Schema erkennen zu können. Die Korrektur kann abhängig von der Fehlerart und der umgebenen Software-Struktur ausgeführt werden.

Notationsverzeichnis

- $G = (E, K)$: Graph mit Eckenmenge E und Kantenmenge K
- SAT (SAT-Solving): Abkürzung für *satisfiability* (englisch für Erfüllbarkeit)
- p_U: Menge aller Pfade, die Ursachen enthalten
- p_R: Menge aller Pfade, die keine Ursachen enthalten
- p: Pfad
- $\omega(p)$: Gewicht von Pfad p
- $k \in p$: Knoten in Pfad p
- $w(k)$: Gewicht von Knoten k
- $\mathbf{U} \in \mathbb{R}_+^{m_U \times n}$: Matrix, welche die Pfade in p_U beschreibt
- $\mathbf{R} \in \mathbb{R}_+^{m_R \times n}$: Matrix, welche die Pfade in p_R beschreibt

- m_U: Anzahl Pfade in p_U
- m_R: Anzahl Pfade in p_R
- n: Anzahl der verschiedenen Knotentypen
- $\mathbf{w} = [w(k_1), \ldots, w(k_n)]^\intercal$: Gewichtevektor für die Knotentypen
- k_i: Knoten vom Typ i
- i, j: Indizes
- $a \geq 0$: untere Grenze des Suchraums, d.h. Infimum vom Gewichtevektor
- $b \geq 0, a < b$: obere Grenze des Suchraums, d.h. Supremum vom Gewichtevektor
- $D = \{\mathbf{w} \in \mathbb{R}^n \mid a \leq w_i \leq b \; \forall \; 1 \leq i \leq n\} \subset \mathbb{R}^n$: Menge der zulässigen Werte für das Optimierungsproblem
- $\mathbf{S} = \begin{bmatrix} -1 & & & \\ & -1 & & \\ & & \ddots & \\ & & & -1 \\ 1 & & & \\ & 1 & & \\ & & \ddots & \\ & & & 1 \end{bmatrix} \in \mathbb{R}^{2n \times n}$:

  Matrix zur Beschreibung der Ungleichung $a \leq w_i \leq b \; \forall \; 1 \leq i \leq n$

- $\mathbf{o} = \begin{bmatrix} -a \\ -a \\ \vdots \\ -a \\ b \\ b \\ \vdots \\ b \end{bmatrix} \in \mathbb{R}^{2n}$:

  Vektor zur Beschreibung der Ungleichung $a \leq w_i \leq b \ \forall\ 1 \leq i \leq n$

- $X \subset \mathbb{R}^n$: Menge der zulässigen Punkte für das Optimierungsproblem
- $\mathbf{x} \in X$: zulässiger Punkt für das Optimierungsproblem
- $\mathbf{g} : \mathbb{R}^n \to \mathbb{R}^m$: Ungleichungsrestriktion im Optimierungsproblem mit $\mathbf{g}(\mathbf{x}) \leq \mathbf{0}$
- $\mathbf{h} : \mathbb{R}^n \to \mathbb{R}^p$: Gleichheitsrestriktion im Optimierungsproblem mit $\mathbf{h}(\mathbf{x}) = \mathbf{0}$
- $\boldsymbol{\lambda} \in \mathbb{R}^m, \boldsymbol{\mu} \in \mathbb{R}^p$: Lagrange-Multiplikatoren
- $q(\boldsymbol{\lambda}, \boldsymbol{\mu})$: duale Funktion
- $\mathfrak{L}(\mathbf{x}, \boldsymbol{\lambda}, \boldsymbol{\mu})$: Lagrange-Funktion
- DP: Dualproblem
- $\tilde{D} = \{\mathbf{w} \in \mathbb{R}^n \mid a < w_i < b \ \forall\ 1 \leq i \leq n\}$: offene, zulässige Menge für das Optimierungsproblem
- $m_k = f(\mathbf{w}^k)$: Folge beim Subgradientenverfahren
- $\mathbf{s}^k$: Subgradient

- $\mathbf{d}^k$: Suchrichtung beim Subgradientenverfahren
- t_k: Schrittweite beim Subgradientenverfahren
- $\mathbf{w}^k$: Lösungsvektor im k-ten Schritt beim Subgradientenverfahren
- f^*: optimaler Funktionswert
- p_A: Menge aller gefundenen Pfade
- $\mathbf{A} \in \mathbb{R}_+^{m \times n}$: Matrix, welche die Pfade p_A beschreibt
- $c \in \mathbb{N}$: Anzahl der höchsten Pfadgewichte

Literaturverzeichnis

1. **Abreu et al. 2007** Abreu, Rui ; Zoeteweij, Peter ; Van Gemund, Arjan J.: On the Accuracy of Spectrum-based Fault Localization. In: Testing: Academic and Industrial Conference Practice and Research Techniques - MUTATION IEEE (Veranst.), 2007, S. 89–98

2. **ADAC 2014a** ADAC: Mehr als 1.500 Rückrufaktionen. `http://www.adac.de/infotestrat/reparatur-pflege-und-wartung/rueckrufe/`, Hersteller: Toyota, Modell: Prius. 2014. – [Eingesehen am 11.11.2014]

3. **ADAC 2014b** ADAC: Mehr als 1.500 Rückrufaktionen. `http://www.adac.de/infotestrat/reparatur-pflege-und-wartung/rueckrufe/`. 2014. – [Eingesehen am 17.11.2014]

4. **Albeck und Albeck 2005** Albeck, Wilfried ; Albeck, Sonja: ...auf dr Sau naus!: Neues vom Saitenwurscht-Äquator. Verlag Albeck, 2005

5. **Alt 2002** Alt, Walter: Nichtlineare Optimierung - Eine Einführung in Theorie, Verfahren und Anwendungen. Vieweg, 2002

6. **Ariane 501 Inquiry Board 1996** ARIANE 501 INQUIRY BOARD: ARIANE 5 - Flight 501 Failure. http://esamultimedia.esa.int/docs/esa-x-1819eng.pdf. 1996. – [Eingesehen am 07.02.2016]

7. **auto motor und sport 2013** AUTO MOTOR UND SPORT: Opel Insignia Rückruf - 61.000 Fahrzeuge mit Softwarefehler. http://www.auto-motor-und-sport.de/news/opel-insignia-rueckruf-61-000-fahrzeuge-mit-softwarefehler-7483237.html. 2013. – [Eingesehen am 11.11.2014]

8. **Automobil Industrie 2012** AUTOMOBIL INDUSTRIE: Elektronik im Auto: Die Entwicklung. http://www.automobil-industrie.vogel.de/elektronik/articles/367049/. 2012. – [Eingesehen am 30.11.2014]

9. **Baier und Katoen 2008** BAIER, Christel ; KATOEN, Joost-Pieter: Principles of Model Checking. Bd. 2. Cambridge, Massachusetts, London, England : MIT press, 2008

10. **Barnat et al. 2013** BARNAT, Jiří ; BRIM, Luboš ; HAVEL, Vojtěch ; HAVLÍČEK, Jan ; KRIHO, Jan ; LENČO, Milan ; ROČKAI, Petr ; ŠTILL, Vladimír ; WEISER, Jiří: DiVinE 3.0 - An Explicit-State Model Checker for Multithreaded C & C++ Programs. In: Computer Aided Verification Springer (Veranst.), 2013, S. 863–868

11. **Berns et al. 2010** BERNS, Karsten ; SCHÜRMANN, Bernd ; TRAPP, Mario: Eingebettete Systeme. Bd. 1. Vieweg+ Teubner, Wiesbaden, 2010

12. **BTC Embedded Systems AG 2014** BTC EMBEDDED SYSTEMS AG: BTC EmbeddedValidator. http://www.btc-es.de/index.php?lang=2&idcatside=5. 2014. – [Eingesehen am 01.04.2014]

13. **Chakraborty et al. 2012** Chakraborty, Samarjit ; Lukasiewycz, Martin ; Buckl, Christian ; Fahmy, Suhaib ; Chang, Naehyuck ; Park, Sangyoung ; Kim, Younghyun ; Leteinturier, Patrick ; Adlkofer, Hans: Embedded Systems and Software Challenges in Electric Vehicles. In: Proceedings of the Conference on Design, Automation and Test in Europe EDA Consortium (Veranst.), 2012, S. 424–429

14. **ChannelPartner 2009** ChannelPartner: Kleine Pannen mit grossen Auswirkungen - Die zehn dramatischsten Softwarefehler. `http://www.channelpartner.de/a/die-zehn-dramatischsten-softwarefehler,277097`. 2009. – [Eingesehen am 10.11.2014]

15. **Chen et al. 2004** Chen, Mike ; Zheng, Alice X. ; Lloyd, Jim ; Jordan, Michael ; Brewer, Eric: Failure Diagnosis Using Decision Trees. In: International Conference on Autonomic Computing IEEE (Veranst.), 2004, S. 36–43

16. **Conrad et al. 2005** Conrad, Mirko ; Fey, Ines ; Grochtmann, Matthias ; Klein, Torsten: Modellbasierte Entwicklung eingebetteter Fahrzeugsoftware bei DaimlerChrysler. In: Informatik-Forschung und Entwicklung 20 (2005), Nr. 1-2, S. 3–10

17. **DER SPIEGEL 1990** DER SPIEGEL: Die programmierte Katastrophe - SPIEGEL-Report über Fehleranfälligkeit von Computersystemen. `http://www.spiegel.de/spiegel/print/d-13501897.html`. 1990. – [Eingesehen am 11.11.2014]

18. **dSPACE 2014** dSPACE: TargetLink: Mehr als ein Seriencode-Generator. `http://www.dspace.com/de/gmb/home/products/sw/pcgs/targetli.cfm`. 2014. – [Eingesehen am 23.11.2014]

19. **dSPACE 2015** dSPACE: Model Compare: Vergleich von Simulink©-, Stateflow©- und TargetLink©-Modellen. `https://www.dspace.com/`

de/gmb/home/products/sw/pcgs/model_compare.cfm. 2015. – [Eingesehen am 14.09.2015]

20. **ELEKTRONIKPRAXIS 2010** ELEKTRONIKPRAXIS: High Integrity Systeme - Mit formaler Verifikation gegen Laufzeitfehler. http://www.elektronikpraxis.vogel.de/themen/embeddedsoftwareengineering/testinstallation/articles/263965/index2.html. 2010. – [Eingesehen am 05.12.2014]

21. **Geiger und Kanzow 2002** Geiger, Carl ; Kanzow, Christian: Theorie und Numerik restringierter Optimierungsaufgaben. Springer-Verlag, 2002

22. **Hangal und Lam 2002** Hangal, Sudheendra ; Lam, Monica S.: Tracking Down Software Bugs Using Automatic Anomaly Detection. In: Proceedings of the 24th international conference on Software engineering ACM (Veranst.), 2002, S. 291–301

23. **heise online 2004** heise online: Softwarefehler plagt Mercedes-Diesel. http://www.heise.de/newsticker/meldung/Softwarefehler-plagt-Mercedes-Diesel-101722.html. 2004. – [Eingesehen am 11.11.2014]

24. **Hildebrandt 2002** Hildebrandt, Stefan: Analysis 1. Springer-Verlag, 2002

25. **IEC - International Electrotechnical Commission** IEC - International Electrotechnical Commission: IEC 61508 Functional Safety. http://www.iec.ch/functionalsafety/. – [Eingesehen am 06.02.2016]

26. **INGENIEUR.de 2001** INGENIEUR.de: Im Auto boomt die Software. http://www.ingenieur.de/Fachbereiche/Elektronik/Im-Auto-boomt-Software. 2001

27. **International Organization for Standardization (ISO) 2012** International Organization for Standardization (ISO): ISO 26262-1:2011. `http://www.iso.org/iso/catalogue_detail?csnumber=43464`. 2012. – [Eingesehen am 15.08.2015]

28. **Jones und Harrold 2005** Jones, James A. ; Harrold, Mary J.: Empirical Evaluation of the Tarantula Automatic Fault-Localization Technique. In: Proceedings of the 20th IEEE/ACM international Conference on Automated Software Engineering ACM (Veranst.), 2005, S. 273–282

29. **Julisch 2003** Julisch, Klaus: Clustering Intrusion Detection Alarms to Support Root Cause Analysis. In: ACM Transactions on Information and System Security (TISSEC) 6 (2003), Nr. 4, S. 443–471

30. **Kästner et al. 2009** Kästner, Daniel ; Ferdinand, Christian ; Wilhelm, Stephan ; Nenova, Stefana ; Honcharova, Olha ; Cousot, Patrick ; Cousot, Radhia ; Feret, Jérôme ; Mauborgne, Laurent ; Miné, Antoine et al.: Astrée: Nachweis der Abwesenheit von Laufzeitfehlern. In: Proc. of Workshop Entwicklung zuverlässiger Software-Systeme (ESS'09), 2009, S. 6

31. **Klaeren und Sperber 2007** Klaeren, Herbert ; Sperber, Michael: Die Macht der Abstraktion: Einführung in die Programmierung. Springer-Verlag, 2007

32. **Kübler et al. 2010** Kübler, Andreas ; Zengler, Christoph ; Küchlin, Wolfgang: Model Counting in Product Configuration. In: Electronic Proceedings in Theoretical Computer Science, 29(LoCoCo) (2010), S. 44–53

33. **Lee 2009** Lee, Che-Rung: Matlab Optimization Toolbox. `http://www.cs.nthu.edu.tw/~cherung/teaching/2009cs5321/link/MatlabOpt.pdf`. 2009. – [Eingesehen am 18.11.2015]

34. **Liu et al. 2015** Liu, Bing ; Lucia, Lucia ; Nejati, Shiva ; Briand, Lionel: Simulink Fault Localization: an Iterative Statistical Debugging Approach. 2015. – Forschungsbericht

35. **Mandau 2009** Mandau, Markus: Computer-Fehler: Die größten Software-Desaster. `http://www.focus.de/digital/computer/chip-exklusiv/tid-14183/computer-fehler-die-groessten-software-desaster_aid_396628.html`. 2009. – [Eingesehen am 10.11.2014]

36. **MathWorks 2014a** MathWorks: Matlab - Die Sprache für technische Berechnungen. `http://de.mathworks.com/products/matlab/`. 2014. – [Eingesehen am 23.12.2014]

37. **MathWorks 2014b** MathWorks: Polyspace Code Prover - Hauptmerkmale. `http://de.mathworks.com/products/polyspace-code-prover/features.html`. 2014. – [Eingesehen am 05.12.2014]

38. **MathWorks 2014c** MathWorks: Simulink - Simulation and Model-Based Design. `http://www.mathworks.com/products/simulink/`. 2014. – [Eingesehen am 23.11.2014]

39. **MathWorks 2014d** MathWorks: Static Analysis with Polyspace Products. `http://www.mathworks.com/products/polyspace/`. 2014. – [Eingesehen am 23.11.2014]

40. **MathWorks 2015a** MathWorks: Choosing a Solver. `http://de.mathworks.com/help/optim/ug/choosing-a-solver.html#bsbqd7i`. 2015. – [Eingesehen am 11.10.2015]

41. **MathWorks 2015b** MathWorks: Contrained Nonlinear Optimization Algorithms - fmincon Active Set Algorithm. `http://de.mathworks.com/help/optim/ug/constrained-nonlinear-optimization-algorithms.html#brnox01`. 2015. – [Eingesehen am 22.07.2015]

42. **MathWorks 2015c** MathWorks: fmincon. http://de.mathworks.com/help/optim/ug/fmincon.html?refresh=true. 2015. – [Eingesehen am 11.10.2015]

43. **Mira Ltd 2013** Mira Ltd: What is MISRA? http://www.misra.org.uk/MISRAHome/WhatisMISRA/tabid/66/Default.aspx. 2013. – [Eingesehen am 15.08.2015]

44. **Post et al. 2008** Post, Hendrik ; Sinz, Carsten ; Kaiser, Alexander ; Gorges, Thomas: Reducing False Positives by Combining Abstract Interpretation and Bounded Model Checking. In: Proceedings of the 2008 23rd IEEE/ACM International Conference on Automated Software Engineering IEEE Computer Society (Veranst.), 2008, S. 188–197

45. **Reder und Egyed 2013** Reder, Alexander ; Egyed, Alexander: Determining the Cause of a Design Model Inconsistency. In: IEEE Transactions on Software Engineering 39 (2013), Nr. 11, S. 1531–1548

46. **safercar.gov 2014** safercar.gov: FLAT FILE COPIES OF NHTSA/ODI DATABASES. http://www-odi.nhtsa.dot.gov/downloads/flatfiles.cfm. 2014. – [Eingesehen am 18.11.2014]

47. **Schneider 2014** Schneider, Johanna: Tracking Down Root Causes of Defects in Simulink Models. In: Proceedings of the 29th ACM/IEEE international Conference on Automated Software Engineering ACM (Veranst.), 2014, S. 599–604

48. **Schrammel et al. 2014** Schrammel, Peter ; Kroening, Daniel ; Brain, Martin ; Martins, Ruben ; Teige, Tino ; Bienmüller, Tom: Incremental Bounded Model Checking for Embedded Software (extended version). In: arXiv preprint arXiv:1409.5872 (2014)

49. **Spiegel Online 2003** Spiegel Online: Tücken der Technik: Thailändischer Minister in BMW gefangen. http://www.spiegel.de/

`panorama/tuecken-der-technik-thailaendischer-minister-in-bmw-gefangen-a-248319.html`. 2003. – [Eingesehen am 11.11.2014]

50. **Spiegel Online 2014** Spiegel Online: Airbag-Fehler beim A4: Audi ruft 850.000 Wagen zurück. `http://www.spiegel.de/auto/aktuell/audi-a4-rueckruf-softwarefehler-kann-fuer-probleme-bei-airbags-sorgen-a-998896.html`. 2014. – [Eingesehen am 11.11.2014]

51. **Werdich 2013** Werdich, Martin: FMEA - Einführung und Moderation: Durch systematische Entwicklung zur übersichtlichen Risikominimierung (inkl. Methoden im Umfeld). Springer-Verlag, 2013

52. **Wikipedia 2014a** Wikipedia: Audi - Modelle seit 1965. `http://de.wikipedia.org/wiki/Audi#Modelle_seit_1965`. 2014. – [Eingesehen am 24.01.2015]

53. **Wikipedia 2014b** Wikipedia: BMW - Modellgeschichte ab 1951. `http://de.wikipedia.org/wiki/BMW_%28Automarke%29#Modellgeschichte_ab_1951`. 2014. – [Eingesehen am 24.01.2015]

54. **Wikipedia 2014c** Wikipedia: Mercedes-Benz-PKW - Modelle seit 1945. `http://de.wikipedia.org/wiki/Mercedes-Benz-PKW#Modelle_seit_1945`. 2014. – [Eingesehen am 24.01.2015]

55. **Wikipedia 2014d** Wikipedia: Toyota - Verkaufsbezeichnungen (Mitteleuropa). `http://de.wikipedia.org/wiki/Toyota#Verkaufsbezeichnungen_.28Mitteleuropa.29`. 2014. – [Eingesehen am 24.01.2015]

56. **Wikipedia 2014e** Wikipedia: Volkswagen - Modelle in Europa. `http://de.wikipedia.org/wiki/Volkswagen#Modelle_in_Europa`. 2014. – Eingesehen am 24.01.2015]

57. **Zeller 2006** Zeller, Andreas: Why Programs Fail - A Guide to Systematic Debugging. dpunkt.verlag, 2006